Proceedings

11th IEEE Workshop on

Real-Time Operating Systems and Software

Seattle, Washington May 18-19, 1994

Sponsored by
IEEE Computer Society Technical Committee on Real-Time Systems

IEEE Computer Society Press

The Institute of Electrical and Electronics Engineers, Inc.

Proceedings

RTOSS '94


RTOSS '94
SEATTLE

Proceedings

11th IEEE Workshop on Real-Time Operating Systems and Software

RTOSS '94

May 18 – 19, 1994
Seattle, Washington

Sponsored by
The IEEE Computer Society Technical Committee on
Real-Time Systems

IEEE Computer Society Press
Los Alamitos, California

Washington • Brussels • Tokyo

IEEE Computer Society Press
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1264

IEEE Computer Society Press Order Number 5710-02
Library of Congress Number 93-81372
IEEE Catalog Number 94TH0639-5
ISBN 0-8186-5710-3 (paper)
ISBN 0-8186-5711-1 (microfiche)

Additional copies may be ordered from:

IEEE Computer Society Press
Customer Service Center
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1264
Tel: +1-714-821-8380
Fax: +1-714-821-4641
Email: cs.books@computer.org

IEEE Service Center
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
Tel: +1-908-981-1393
Fax: +1-908-981-9667

IEEE Computer Society
13, Avenue de l'Aquilon
B-1200 Brussels
BELGIUM
Tel: +32-2-770-2198
Fax: +32-2-770-8505

IEEE Computer Society
Ooshima Building
2-19-1 Minami-Aoyama
Minato-ku, Tokyo 107
JAPAN
Tel: +81-3-3408-3118
Fax: +81-3-3408-3553

Editorial production by Bob Werner
Cover art design and production by Michael Nomura
Printed in the United States of America by Braun-Brumfield, Inc.

The Institute of Electrical and Electronics Engineers, Inc.

Contents

Session IV: Timing Analysis
Chair: Stuart Faulk, SPC

Session V: Scheduling II
Chair: Hide Tokuda, CMU

Panel: Real-Time Bench Marks
Chair: Karsten Schwan, Georgia Tech

Session VI: Operating Systems II
Chair: Keith Marzullo, UCSD

Session VII: Concurrency Control
Chair: Vic Wolfe, U. Rhode Island

Panel: If Scheduling Is So Important, Why Aren't Folks Beating a Path to Our Door?
Chair: Kevin Jeffay, University of North Carolina

Chairs' Message

The IEEE Workshop on Real-Time Operating Systems and Software is a forum that covers recent advances in real-time computing — a field that is becoming an essential part of computer science and engineering. It brings together practitioners and researchers from academia, industry, and government, to explore the best current ideas on real-time software and operating systems, and to evaluate the maturity and directions of real-time system technology. As the demand for the functionalities and reliabilities of real-time systems continue to grow, our intellectual and engineering abilities are being challenged to come up with practical solutions to the problems faced in design and development of complex real-time systems.

The interest in this important topic is confirmed by the high number of quality submissions. Following the tradition of previous RTOSS workshops, parallel sessions are avoided in order to give participants the opportunity to be involved in interactions with speakers and panelists, and to exchange opinions with all other participants. As a consequence, many good position papers had to be rejected.

The technical program covers a wide range of issues, such as scheduling, operating systems, communications, timing analysis, system design, concurrency control, and formal methods. Besides the various sessions, the program includes three panel sessions to address important issues on real-time programming languages, education, and real-time scheduling. In addition, Nancy Leveson from the University of Washington will deliver an invited talk on software safety.

Many people worked hard to make this year's RTOSS workshop a success. The Program Committee members carefully reviewed and discussed every submitted paper, and made the difficult decisions on which papers to accept. We also would like to thank the authors of all the submitted papers. Special thanks go to Alicen Smith for managing the administrative activities, and Bob Werner of the IEEE Computer Society for the publication of this proceedings. Finally, we are grateful to the IEEE Computer Society Technical Committee on Real-Time Systems, the Office of Naval Research, and the Departments of Computer Science at the University of Virginia and the University of Washington.

Welcome to Seattle!

Sang H. Son, General Chair
University of Virginia

Alan Shaw, Program Chair
University of Washington

Program Committee

Eleventh IEEE Workshop on
Real-Time Operating Systems and Software

General Chair

Sang H. Son
Department of Computer Science
University of Virginia
Charlottesville, VA 22903 USA
phone: (804) 982-2205
fax: (804) 982-2214
son@virginia.edu

Program Chair

Alan C. Shaw
Department of Computer Science and Engineering FR-35
University of Washington
Seattle, WA 98195 USA
phone: (206) 543-9298
fax: (206) 543-2969
shaw@cs.washington.edu

Program Committee

Ted Baker, Florida State University
Stuart Faulk, Software Productivity Consortium
Mike Jones, Microsoft Corporation
Luqi, U.S. Naval Postgraduate School
Keith Marzullo, UC San Diego
Karsten Schwan, Georgia Tech
Hideyuki Tokuda, Carnegie Mellon University
Wei Zhao, Texas A&M University

Session I: Operating Systems I

Chair: Karsten Schwan
Georgia Tech.

Predictable Spin Lock Algorithms with Preemption

Hiroaki Takada and Ken Sakamura

Department of Information Science,
Faculty of Science, University of Tokyo
7-3-1, Hongo, Bunkyo-ku, Tokyo 113, Japan

Abstract

Both predictable interprocessor synchronization and fast interrupt response are required for real-time systems constructed using asymmetric shared-memory multiprocessors. This paper points out the problem that conventional spin lock algorithms cannot satisfy both requirements at the same time. To solve this problem, we have proposed an algorithm which is an extension of queueing spin locks modified to be preemptable for servicing interrupts [1]. In this paper, we propose an improved algorithm that minimizes the recovering overhead from an interrupt service. We also demonstrate that the proposed algorithms have required properties through performance measurement.

1 Introduction

In many applications of high performance real-time systems, a large number of external devices such as sensors, actuators, and network controllers are connected to a system and the system is required to respond to the external events from the devices within predefined and usually short time-bounds. To meet this requirement, asymmetric multiprocessors in which each device is handled by a fixed processor are often adopted.

In order to realize real-time systems using shared-memory multiprocessors, predictable interprocessor synchronization mechanisms are of primary importance. In addition to adopting a real-time scheduling algorithm with resource constraints or a real-time synchronization protocol, the execution time of the underlying mutual exclusion mechanism using spin locks must be bounded[1].

In asymmetric shared-memory multiprocessors, each processor is required to achieve fast and predictable response to interrupt requests, because external events are notified to each processor in the form of interrupts. However, each processor cannot respond to external interrupts in short latency with conventional bounded spin lock algorithms.

To solve this problem, we have proposed an algorithm which is an extension of queueing spin locks modified to be preemptable for servicing interrupts [1]. With the algorithm, an upper bound on the time to acquire and release an interprocessor lock can be given when no interrupt request occurs, and fast response to interrupt requests is achieved. However, the algorithm has a shortcoming that a processor possibly has to re-execute the lock acquiring routine from the *beginning* after it services an interrupt request. In schedulability analysis, this re-execution overhead must be added to the interrupt service time.

In this paper, we propose an improved algorithm that minimizes this overhead. We also demonstrate that the proposed algorithms have required properties through performance measurement.

2 Spin locks and interrupt latency

In this paper, we assume that atomic read-modify-write operations on a single word of shared memory (e.g. test_and_set, fetch_and_store (swap), fetch_and_add, and compare_and_swap) are supported in hardware.

In order to bound the time until a processor acquires an interprocessor lock, the duration that each processor holds the lock must be bounded as well as the number of contending processors that the processor must wait for. The latter condition can be met with ticket locks or queueing locks [2], with which the turn that a processor acquires a lock is determined when it begins waiting for the lock. To satisfy the former condition, the relationship with interrupt services must be considered.

In asymmetric multiprocessor systems, interrupt services for external devices are requested for each processor. When multiple devices are connected to a processor, interrupt requests from them are usually raised independently and the maximum time to service all of the requests becomes unbounded or very long. Consequently, in order to give a practical bound on the duration that a processor holds a lock, interrupt services should be inhibited for that duration.

On the other hand, in order to realize a system with fast response to external events, each processor must be able to service external interrupts with short latency time. Particularly, when the scalability of the system is an impor-

[1] We assume that the access time of the shared bus (or interconnection network) is bounded in this paper.

0-8186-5710-3/94 $3.00

tant issue, the worst-case interrupt latency should be given independently of the number of processors in the system.

Here a problem arises in deciding whether interrupts should be disabled first or an interprocessor lock should be acquired first. When acquiring an interprocessor lock precedes disabling interrupts, interrupts may be serviced while the processor holds the lock, and the condition that interrupt services should be inhibited while a processor holds a lock is not satisfied. If acquiring a lock follows disabling interrupts, on the other hand, the interrupt mask time includes the time to acquire the lock and its upper bound heavily depends on the number of processors.

One method to solve this problem is the following. The processor first disables interrupts and tries to acquire the lock. If it fails to acquire the lock, the processor probes interrupt requests before it retries to acquire the lock. When interrupt requests are detected, it suspends trying to acquire the lock, enables interrupts, and services them.

Test-and-set locks can be extended easily with this method. Ticket locks and queueing locks, on the other hand, cannot be extended similarly.

3 Queueing locks with preemption

In all spin lock algorithms that can give an upper bound on the time until a processor acquires a lock, a processor modifies some shared variable and reserves its turn to acquire the lock when it begins waiting for the lock. When its turn comes, the lock is passed to the processor by another. If the processor simply branches to an interrupt handler while waiting for the lock, it cannot begin to execute the critical section immediately after the lock is passed to the processor, and makes the contending processors wait wastefully until the interrupt service is finished.

Consequently, when a processor begins to service interrupts while waiting for a lock, it must inform others that it is servicing interrupts and should not be passed the lock. The processor trying to release the lock checks if the succeeding processor is servicing interrupts. If the succeeding one is found to be servicing interrupts, its turn to acquire the lock is canceled or deferred, and the lock is passed to the next in line.

Original algorithm

We have applied the above scheme to the MCS lock, a list-based queueing lock algorithm [2], and proposed a queueing lock algorithm with preemption [1]. Some other spin lock algorithms can be extended similarly. Recently, R. W. Wisniewski et al. have proposed a similar algorithm for improving the average performance of multiprogrammed (non-real-time) systems [3]. Craig's algorithm can also support the same preemption scheme [4].

In the algorithm, if the processor trying to release the lock (P_0) finds that the succeeding processor (P_1) is servicing interrupts, P_0 dequeues P_1 from the waiting queue and passes the lock to a successor of P_1. When only P_1 is waiting for the lock, P_0 makes the waiting queue empty. P_0 informs P_1 that P_1 is dequeued using a shared variable. When P_1 finishes the interrupt service, it checks whether it has been dequeued during the interrupt service or not. If it has been dequeued, it re-executes the lock acquiring routine from the beginning. Otherwise, it resumes waiting for the lock.

When a processor is dequeued and re-executes the lock-acquiring routine, the waiting time after the processor first links itself to the queue until it branches to the interrupt handler is wasted. When the schedulability of the system is analyzed, this re-execution overhead should be added to the interrupt service time. Below, we present an improved algorithm which is devised to reduce this overhead.

Improved algorithm

The re-execution overhead can be reduced with the following method. When the processor releasing the lock (P_0) finds that the succeeding processor (P_1) is servicing interrupts, P_0 leaves P_1 in the waiting queue instead of dequeueing it. P_0 removes the processor to which to pass the lock from the queue using the method adopted in the prioritized queueing spin lock appeared in [5]. When P_1 finishes interrupt services, it simply resumes waiting for the lock in its original position. Therefore, the overhead which must be added to the interrupt service time in schedulability analysis is minimized.

A difficulty occurs when all processors in the waiting queue are servicing interrupts. To handle this situation, a global lock flag is introduced. If the processor trying to release the lock finds that all processors in the queue are servicing interrupts, it sets the global lock flag. A processor returning from interrupt services tries to get the global lock with the same method as with test-and-set locks. If it succeeds getting the lock, it removes itself from the waiting queue. As the processor needs to know the top processor in the queue to remove itself, the processor releasing the global lock must pass the information in some shared variable. It is also necessary for a processor to check the global lock flag once, after it links itself at the end of the queue, because it is possible that all the processors in the queue are servicing interrupts and the global lock is set.

Pseudo-code for the improved algorithm appears in Fig. 1 and 2. In these figures, the keyword **shared** indicates that only one instance of the variable is allocated and shared in the system. Other variables are allocated for each processor and located in its local memory. The right hand side of the **and** operator is assumed to be evaluated only if its left hand side is true. **Fetch_and_store** reads the memory addressed by the first parameter, returns the contents of the memory as its value, and atomically writes the second parameter to the memory. **CAS**, the abbreviation of compare_and_swap, first reads the memory pointed to by the first parameter and compares its contents

```
type qnode = record
    next, prev: pointer to qnode;
    locked: (Released, Locked, Preempted, Dequeueing)
end;
type lock = record
    last: pointer to qnode;
    glock: pointer to qnode
end;

// global shared data.
shared var L: lock;
// L.last and L.glock are initialized to NIL.

procedure dequeue(entry, pred, top: pointer to qnode)
    var succ: pointer to qnode;
    succ := entry→next;
    if succ = NIL then
        pred→next := NIL;
        if CAS(&(L.last), entry, pred) then goto release end;
#       repeat succ := entry→next until succ ≠ NIL
    end;
    pred→next := succ;
    succ→prev := pred;
  release:
    entry→next := top;
    entry→locked := Released
end;
```

Fig. 1: Improved algorithm (1)

with the second parameter. If they are equal, the function writes the third parameter to the memory atomically and returns true. Otherwise, it returns false.

In this pseudo-code, the **glock** field of **L** serves both as the global lock flag and as the variable to pass the top processor of the waiting queue. An exponential backoff scheme is adopted to get the global lock in this code to reduce the number of shared-bus transactions. Two constant parameters α and β should be tuned for each target hardware and application.

Though there are two non-local spins (marked with #) in this pseudo-code, both of them continue during the transient state afer another processor writes the pointer to its queue node to **L.last** (successful execution of the fetch_and_store operation marked with ①) and until it writes non-NIL value to the **next** field of its predecessor (marked with ②), and their effect is not significant.

We have adopted the MCS lock as the base algorithm in this section. The FIFO version of Craig's algorithm [4] can be extended similarly.

4 Performance evaluation

The effectiveness of the two queueing spin lock algorithms with preemption, the original one in [1] (called QL/P1, in this section) and the improved one presented in Fig. 1 and 2 (QL/P2), are examined through performance evaluation. The performance of the algorithms is compared with the MCS lock without inhibiting interrupts (QL/ei), the MCS lock during interrupts inhibited (QL/di),

```
// local data (allocated for each processor).
var I: qnode;
var pred, succ, top: pointer to qnode;
var interval, i: integer;

I.next := NIL;
disable_interrupts;
① pred := fetch_and_store(&(L.last), &I);
if pred = NIL then goto acquired end;
// enqueue myself.
I.prev := pred;
I.locked := Locked;
② pred→next := &I;
i := 1;        // check the global lock once.
interval := ∞;        // never expires.
while (I.locked ≠ Released) do
    if interrupt_requested and
            CAS(&(I.locked), Locked, Preempted) then
        enable_interrupts;
        // interrupt service.
        disable_interrupts;
        I.locked := Locked;
        i := 1;
        interval := α
    end;
    i := i - 1;
    if i = 0 then
        // check the global lock and try to get if it is set.
        top := L.glock;
        if top ≠ NIL and CAS(&(L.glock), top, NIL) then
            if top ≠ &I then dequeue(&I, I.prev, top) end;
            goto acquired
        end;
        i := interval;
        interval := interval × β
    end
end;
acquired:
//
// critical section.
//
succ := I.next;
if succ = NIL then
    // try to make the queue empty.
    if CAS(&(L.last), &I, NIL) then goto exit end;
    repeat succ := I.next until succ ≠ NIL
end;
// try to pass the lock to the successor.
if CAS(&(succ→locked), Locked, Released) then goto exit end;
top := succ;
repeat
    pred := succ;
    succ := pred→next;
    if succ = NIL then
        // set the global lock.
        L.glock := top;
        // check if pred is really the last processor.
        if L.last = pred then goto exit end;
        // try to withdraw the global lock.
        if ¬CAS(&(L.glock), top, NIL) then goto exit end;
#       repeat succ := pred→next until succ ≠ NIL
    end;
until CAS(&(succ→locked), Locked, Dequeueing);
dequeue(succ, pred, top);
exit:
enable_interrupts;
```

Fig. 2: Improved algorithm (2)

```
for i := 1 to NoLoop do
① acquire_lock_and_disable_interrupts;
   //
   // critical section.
   //
   release_lock;
② enable_interrupts;
   random_delay
end;
```

Fig. 3: Measurement program skeleton

and the test-and-set lock with preemption with constant delay (T&S/P)[2].

Evaluation environment

We have used a shared-bus multiprocessor system for the evaluation. The shared bus is based on the VME-bus specification, and each processor node consists of a 20 MHz GMICRO/200 microprocessor, which is rated at approximately 10 MIPS, 1 MB of local memory, and some I/O interfaces. The local memory can be accessed from other processors through the shared bus. No cache memory is equipped. The program code and the data area for each processor are placed in the local memory of the processor. Global shared data (e.g. L in Fig. 1) is placed in the local memory of the master processor, which does not execute spin locks.

The GMICRO/200 microprocessor supports the compare_and_swap instruction but not fetch_and_store. In our experiments, the fetch_and_store operation was emulated using the compare_and_swap instruction and a retry loop. As the VMEbus has only four pairs of bus request/grant lines, processors are classified into four classes by the bus request line they use. The round-robin arbitration scheme is adopted among classes and the static priority scheme is applied among processors belonging to a same class.

Measurement method

Each processor executes the code presented in Fig. 3 while periodic interrupt requests are raised on the processor. The execution time of a critical region (the region between ① and ② in Fig. 3) is measured for each execution, and its distributions when the processor services no interrupt request during the region and when it services an interrupt are collected. The interrupt latency is also measured for each interrupt service and its distribution is obtained.

Inside the critical section, a processor accesses the shared bus some number of times (for making the effect of bus traffic explicit) and waits for a while using empty loops. Without spin locks, the execution time of the critical region is about 40 μs including some overhead for obtaining the execution time of the region. In order to change timing conditions, each processor waits for a random time before it re-enters the critical region (*random_delay* in Fig. 3). The average time of the random delay is about 40 μs.

Empty loops are also included in the interrupt handler in addition to the routine for obtaining interrupt latency time. The total execution time of the interrupt handler is about 80 μs. The period of interrupt requests is about 5 ms. The exact length of the period is varied in 0–2% for each processor.

Performance metric

In real-time systems, the effectiveness of algorithms should not be evaluated with their average performance but with their worst-case execution (or response) times. However, in the case of spin lock algorithms, worst-case times cannot be obtained through experiments because of unavoidable non-determinism in multiprocessor systems. Therefore, in place of worst-case times, we have adopted p-reliable times, the time within which a processor finishes executing a critical region (or responds to an interrupt request) with probability p, as a performance metric. In the following section, we show the evaluation results when p is 0.999 (i.e. 99.9%).

Evaluation results

Fig. 4 presents the 99.9%-reliable execution time of the critical region (when no interrupt is serviced on the processor during the region) as the number of processors is increased from one to eight. With QL/P1 and QL/P2, the execution time of the critical region increases linearly with the number of processors, and the algorithms are found to be scalable. QL/ei exhibits poorer performance because preceding processors service interrupt requests during the critical region.

In Fig. 5, the interrupt latency time is nearly independent of the number of processors with QL/P1 and QL/P2. With QL/di on the contrary, the interrupt latency becomes long as the number of processors increases.

From these observations, it is demonstrated that QL/P1 and QL/P2 can give a practical upper bound on the time to acquire and release an interprocessor lock while achieving fast response to interrupt requests. The other algorithms cannot satisfy these two requirements at the same time.

The overall performance of QL/P2 is a little worse than QL/P1, because the number of shared-bus transactions is large with QL/P2 and because doubly linked queue is necessary. The advantage of QL/P2 appears in Fig. 6 which presents the 99.9%-reliable execution time of the critical region when an interrupt is serviced during the region. When the number of processors is large, the recovering overhead from interrupt services is much smaller in QL/P2 than in QL/P1.

[2]Past studies show that a test-and-set lock has good scalability with exponential backoff [2]. However, because the lock acquisition time varies widely with exponential backoff, it is inappropriate for real-time systems. This conjecture was also confirmed through our experiments.

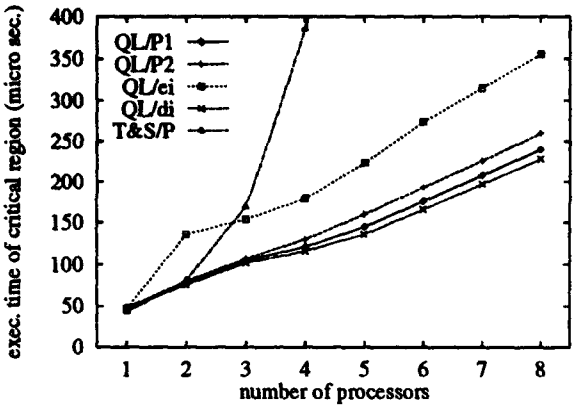


Fig. 4: 99.9%-reliable exec. time of critical region (when no interrupt is serviced)

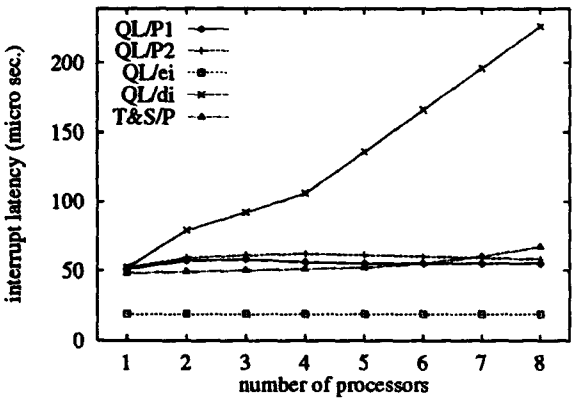


Fig. 5: 99.9%-reliable interrupt latency

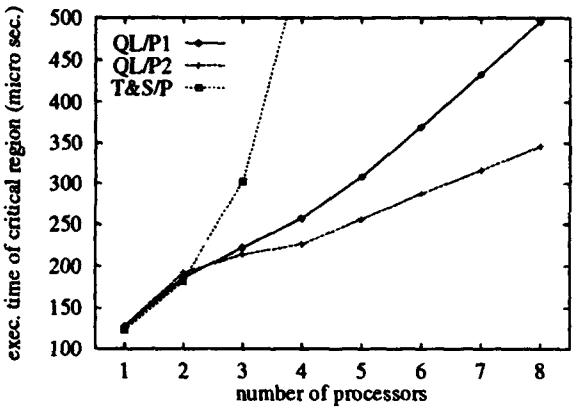


Fig. 6: 99.9%-reliable exec. time of critical region (when an interrupt is serviced)

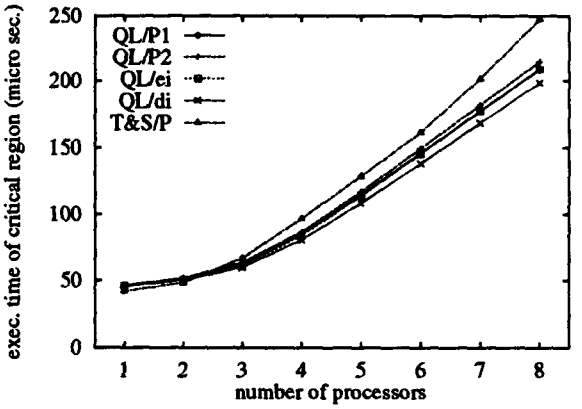


Fig. 7: Average exec. time of critical region

Finally, in order to examine the average performance of the algorithms, we present the average execution time of the critical region (when no interrupt is serviced during the region) in Fig. 7.

5 Conclusion

Conventional spin lock algorithms cannot satisfy two important requirements for real-time systems using asymmetric shared-memory multiprocessors, predictable spin locks and fast interrupt response, at the same time. In this paper, we propose a improved spin lock algorithm that can give an upper bound on the time to acquire and release an interprocessor lock while realizing fast response to interrupt requests. To evaluate their effectiveness, we have measured their performance through experiments and confirmed that the algorithms have the required properties.

We are currently designing a real-time kernel specification called ITRON-MP and implementing it experimentally [6]. It remains as a future work to adopt the algorithms in the implementation and to evaluate the algorithms in real applications.

References

[1] H. Takada and K. Sakamura, "A bounded spin lock algorithm with preemption," Tech. Rep. 93-2, Department of Information Science, University of Tokyo, July 1993.

[2] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. on Computer Systems*, vol. 9, pp. 21–65, Feb. 1991.

[3] R. W. Wisniewski, L. Kontothanassis, and M. L. Scott, "Scalable spin locks for multiprogrammed systems," Tech. Rep. TR454, Computer Science Department, University of Rochester, Apr. 1993.

[4] T. S. Craig, "Queuing spin lock algorithms to support timing predictability," in *Proc. Real-Time Systems Symposium*, pp. 148–157, Dec. 1993.

[5] E. P. Markatos, "Multiprocessor synchronization primitives with priorities," in *Proc. of the IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.

[6] H. Takada and K. Sakamura, "ITRON-MP: An adaptive real-time kernel specification for shared-memory multiprocessor systems," *IEEE Micro*, vol. 11, pp. 24–27,78–85, Aug. 1991.

User-Level Real-Time Threads

Shuichi Oikawa
Faculty of Environmental Information
Keio University

Hideyuki Tokuda
School of Computer Science
Carnegie Mellon University

Abstract

Continuous-media applications require more efficient and flexible support from real-time threads than traditional real-time systems. It includes functionalities such as the dynamic management of thread attributes and the support of multiple thread models. In this paper, we will describe the design and implementation of user-level real-time threads on the RT-Mach micro kernel. Since they are implemented at user-level, both of the fast management of thread attributes and the support of multiple thread models are possible.

1 Introduction

Continuous-media applications require more efficient and flexible support from real-time threads than traditional real-time systems [4, 12, 13]. The "flexible support" includes the following two functionalities:

- the dynamic management of thread attributes,
- the support of multiple thread models.

The dynamic management of thread attributes is necessary because system resource utilization in workstations and network environments is changing every minute. Timing attributes, such as start time, deadline and period, are parts of thread attributes. For example, if there are too many threads for a system to satisfy their timing requirements, some threads may be able to run more infrequently or with shorter execution time. As another example, if network traffic is crowded and an application cannot receive data at the expected rate, threads of the application should change their behavior to follow the rate of data received.

Ability to support multiple thread models is also important. Since there is no standard way to implement continuous-media applications, programmers may be able to choose one of the existing thread packages or may want to create a new one. For instance, one programmer finds it is useful to use a periodic thread to process continuous data, while another programmer would like to use threads which have their start time and deadline and to create a new thread for each data chunk.

Our goal is to realize high performance user-level real-time threads because only user-level real-time threads can achieve the above functionalities. Since they are implemented at user-level, both of the fast management of thread attributes and the support of multiple thread models are possible. The next section describes the previous work. Section 3 discusses design issues of user-level real-time threads, and Section 4 proposes a software architecture for user-level real-time threads. Section 5 describes the current status with some performance figures, and Section 6 gives the conclusion.

2 Previous Work

Real-time threads have been developed as kernel entities. Existing real-time kernels, such as ARTS [10] and RT-Mach [11], realize their real-time threads as kernel-provided threads. Since threads are implemented in the kernel, primitives like real-time synchronization and functions to set thread attributes are also implemented in the kernel. Thus, the thread operations cost so expensive that the performance is sometimes unacceptable for dynamic environments requiring the dynamic management of thread attributes [13].[1]

First-class user-level threads were developed to solve scheduling problems occurred in user-level threads environments where an entire task is blocked when a user-level thread is blocked in the kernel. Scheduler Activations [1] and the first-class user-level threads of the Psyche operating system [7] provide the mechanisms to avoid the above problem. Both of them are implemented on the parallel computers to exploit the ability of parallelism of the underling hardware. Thus, they have no functionality to manage timing attributes of threads.

Split-level scheduling [4] provides user-level real-time threads through the shared user/kernel structures with the

[1] In our previous experience with ARTS [8], context switching in the same address space costs 3μsec for user-level threads and 26μsec for kernel-provided threads. Synchronization costs 9μsec for user-level threads and 46μsec for kernel-provided threads. Since the dynamic management of thread attributes introduces many operations in the same address space (described in Section 4.3), this feature is useful.

0-8186-5710-3/94 $3.00 © 1994 IEEE

split kernel-level and user-level schedulers. This is implemented on a uniprocessor, and shared memory is extensively used to pass information between a user-level scheduler and the kernel. Each user-level thread has its logical arrival time and deadline, and the threads are scheduled by the deadline/workahead scheduling policy based on their timing attributes. Split-level scheduling proposes a new mechanism for asynchronous communication to avoid threads blocked in the kernel. Since the split-level scheduling was developed to handle continuous-media efficiently, its goal is similar to ours. However, it does not have a notion of dynamic rebinding of timing attributes. The timing attributes of threads is managed by the split kernel-level and user-level schedulers cooperatively, while our user-level real-time threads manage timing attributes of threads using a timer which is a separate instance from a thread. This feature increases the flexibility of user-level schedulers.

3 Design Issues

User-level real-time threads must be treated as first-class user-level threads [1, 4, 7] since user-level real-time threads need to be scheduled as correctly as kernel-provided threads. In this section, we first describe the design decisions to implement first-class user-level threads. Then, several design issues are discussed.

3.1 First-Class User-Level Threads

Mechanisms proposed by previous implementations of first-class user-level threads [1, 4, 7] were examined. Then, the following mechanisms were chosen for implementing our user-level real-time thread model.

Upcall: The kernel has to notify a user-level scheduler of events which were occurred in the kernel and affect a scheduling decision. The kernel upcalls a user-level scheduler, and the user-level scheduler processes events and choose the next thread to run. This mechanism is used on all implementations of first-class user-level threads while they call it differently.

Shared kernel/user data structures: There are two different approaches to pass events to a user-level scheduler. Shared kernel/user data structures are used for first-class user-level threads [7] and split-level scheduling [4]. Scheduler activations [1] upcall different entry points of a user-level scheduler each of which is provided for the corresponding type of events. We chose to use shared kernel/user data structures since they can be used to pass information of threads from user-level schedulers to the kernel, such as priorities and timing attributes. It can also provide a simple way to pass events asynchronously.

Creation of a new virtual processor: Scheduler activations [1] create a new virtual processor when the current one is blocked in the kernel, but others do not do so. We chose to create a new virtual processor. There are two main reasons for this decision. One reason is that our platform, RT-Mach [11], requires it. Virtual processors are implemented using kernel-provided threads. Since there are many places where a thread structure is referenced in the kernel, it is too hard to modify them to cope with a user-level thread. Another reason is that the number of interactions between a user-level scheduler and the kernel can be reduced. For example, when a thread is unblocked in the kernel, the event is notified to a user-level scheduler. If the user-level scheduler decides to run the unblocked thread, it issues a system primitive to resume it. We can avoid such a heavy interaction if the current virtual processor is preserved and a new one is used to upcall a user-level scheduler.

3.2 Dynamic Creation of Virtual Processors

The dynamic creation of a new virtual processor sometimes takes a long time, and it can be a source of the unpredictability. If there is an extra virtual processor which is not in use, it can be used instead of the current one. Then, the dynamic creation is not necessary. Therefore, when a user-level scheduler is initializing its status, it asks the kernel to create several kernel-provided threads. Those threads are maintained in the kernel, and are used later as virtual processors when a running virtual processor is blocked in the kernel.

The number of kernel-provided threads created at initialization is fixed. If all of them are used and blocked in the kernel, the kernel needs to create a new virtual processor dynamically or just leaves it blocked. For hard real-time applications, the behaviors are analyzed and the necessary number of virtual processors is found. For soft real-time applications, the dynamic management of the number of virtual processors is necessary.

3.3 Priority Consistency

User-level real-time threads are managed and scheduled by a user-level scheduler, while virtual processors are scheduled by the kernel-level scheduler. User-level real-time threads and virtual processors have their own priority data. Thus, they are managed independently. Since user-level threads are multiplexed on a virtual processor, the priority of the current user-level thread must be reflected to the priority of its virtual processor to schedule the virtual processor correctly.

The problem which arises here is that the current priority which needs to be reflected to the virtual processor changes independently of the kernel because user-level

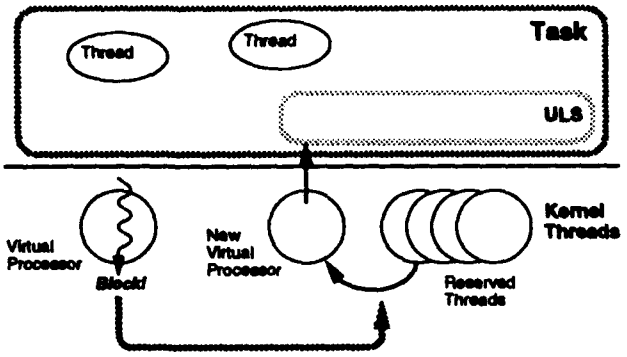


Figure 1: Blocking Thread in the Kernel

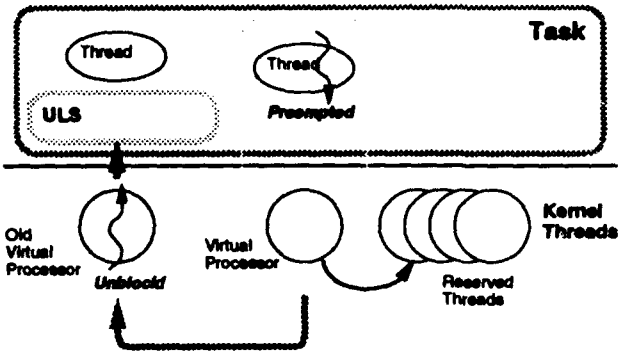


Figure 2: Unblocking Thread in the Kernel

threads switches at user-level. Therefore, a mechanism which makes the priority data of a virtual processor updated is necessary.

3.4 Timing Management

User-level real-time threads also have timing attributes such as a start time, a deadline and so on. Usually, the timing management is done in the kernel using a clock device which interrupts the kernel at intervals of very short period.[2] Since user-level real-time threads are managed by a user-level scheduler, a user-level scheduler needs to manage their timing attributes. This requires for a user-level scheduler the close cooperation with the kernel.

A user-level scheduler needs to tell the kernel when it would like to be notified. Since the dynamic management of thread attribute requires fast rebinding of the timing attribute, system primitives cost too much to do so. Thus, a shared kernel/user data structure is used to share such information. A user-level scheduler maintains timing data in it, and the kernel checks it. If the time which a user-level scheduler needs a notification comes, then the kernel sends an event to it.

4 Software Architecture

In this section, we first describe how virtual processors are used and interact with user-level real-time threads. Then, mechanisms for user-level timers and the dynamic management of timing attributes are discussed.

4.1 Virtual Processors

There are the following three types of virtual processors:

- A *current virtual processor* is currently executing user-level threads in an address space. Only this type of virtual processors can run at user-level.
- A *kernel virtual processor* is attached to a specific user-level thread, which is blocked in the kernel. It executes only in the kernel because user-level threads running at user-level must be multiplexed on the current virtual processor.
- A *reserved virtual processor* is waiting to become a current one. One of them is used when a current one is blocked in the kernel.

When a user-level thread is blocked in the kernel, the current virtual processor, which is executing the blocked thread, becomes a kernel virtual processor. Then, one of reserved virtual processors is taken from the list, and becomes the current virtual processor. Finally, the new current virtual processor upcalls the user-level scheduler. (See Figure 1.)

When a kernel virtual processor is unblocked, it is scheduled by the kernel-level scheduler independently of the current virtual processor. When a kernel virtual processor is about to exit the kernel, it passes two execution contexts to the user-level scheduler. One is for the user-level thread on the kernel virtual processor. Another is for the user-level thread on the current virtual processor, which is preempted by the kernel virtual processor. Then, the current virtual processor is linked in the list of reserved virtual processors, and the kernel virtual processor becomes the new current virtual processor. Finally, the new current virtual processor upcalls the user-level scheduler. (See Figure 2.)

4.1.1 Priority Update

To make the priority data of the current virtual processor consistent with the priority of the current user-level thread, it is updated in the following cases:

- when an interrupt is occurred,

[2] 10ms is a very common value for current workstations.

- when the kernel-level scheduler is invoked,
- when a user-level thread waked up by a timer has a higher priority than the current virtual processor.

At each interrupt, The priority data of the current user-level thread is copied to the current virtual processor in the current task. Then, the kernel checks if the current virtual processor has the highest priority. If it doesn't, the kernel invokes the highest priority kernel-provided thread.

When the kernel-level scheduler is invoked, the priority data of the current user-level thread is copied if the current kernel-provided thread is a virtual processor. Then, we can avoid the priority inconsistency if a user-level thread is switched after an interrupt.

We discuss priority update which is necessary when a user-level thread waked up by a timer in Section 4.2.2.

4.2 Timer

In RT-Mach, a kernel-provided timer called RT-Mach Timer [9] is already implemented in the kernel for kernel-provided real-time threads. To use it for user-level real-time threads, several modifications are necessary to interact with user-level schedulers.

It is possible to use kernel-provided timers with a few modifications if one timer is used for each single user-level real-time thread as kernel-provided real-time threads. This scheme, however, causes a lot of kernel interventions since each operation on a timer is required to issue a system primitive. Semantics of a timer is also limited since it is implemented in the kernel. Then, it makes difficult to achieve our goals.

In our architecture, a user-level scheduler employs a single kernel-provided timer only for notification, and manages *user-level timers* to decide what is necessary to do when notified.

4.2.1 User-Level Timer

User-level timers are managed by a user-level scheduler. A user-level timer provides a kernel-provided timer with the time and the priority data. The time specifies when the user-level scheduler would like to get a notification. The priority data is used by the kernel to update the priority data of the current virtual processor. A kernel-provided timer uses the above data of user-level timers, then decides when it notifies the user-level scheduler. Since data of user-level timers is written by a user-level scheduler and read by a kernel-provided timer, it needs to be placed in a shared kernel/user data structure.

Decoupling user-level threads and timers makes it possible to support multiple thread models. The kernel just notifies a user-level scheduler when it needs a notification. This mechanism does not assume any model. Thus, user-level schedulers can interpret and use notifications as they wish.

4.2.2 Thread Wakeup by Timer

When a user-level thread which is waked up by a timer has the highest priority, the current thread is preempted and the waked up thread must be invoked. This is the same case as when a user-level thread is unblocked in the kernel. Thus, the kernel does the same operations on threads. If a waked up thread does not have the highest priority, the kernel just notifies the event to the user-level scheduler.

4.3 Dynamic Management of Timing Attribute

The dynamic management of thread timing attributes is archived using deadline handlers and dynamic rebinding of thread timing attributes [14].

A *deadline handler* is an independent thread which is attached to a real-time thread. The deadline handler of a real-time thread is invoked when the deadline of the real-time thread is missed. In the deadline handler, it can resume the real-time thread to continue the rest of work although the deadline is missed, or it can abort the invocation if it is meaningless to continue the work after the deadline.

When a system becomes overloaded and deadlines of real-time threads start being missed, their deadline handlers are invoked. In such case, they can rebind the timing attributes of the threads dynamically to reduce the system load. Dynamic rebinding of thread timing attributes resets timing attributes of a real-time thread, such as a period and a deadline, to new values. The new values become valid from the next invocation.

The above operations are all processed at user-level. Thus, user-level real-time threads can achieve much higher performance than kernel-provided real-time threads since kernel interventions are not involved. A deadline handler is an example of mechanisms for the dynamic management. It is very easy to add new features to a user-level scheduler.

5 Current Status

We are currently implementing user-level real-time threads on RT-Mach [11]. As our first implementation of user-level thread packages, we decided to modify C-Threads package [3]. Since our implementation is upper compatible with the original C-Threads package, applications using C-Threads can also benefit from high performance of first-class user-level threads.

Table 1 shows the performance of signal/wait primitives

RTC-Threads (user-level)	C-Threads (user-level)	RT Threads (kernel-provided)
25μsec	38μsec	170μsec

Table 1: Signal/Wait Primitives

null function call	null system call (trap)	null system call (via MIG)
0.8μsec	5μsec	72μsec

Table 2: Basic Operations Performance

of our real-time version of C-Threads (RTC-Threads),[3] original C-Threads and kernel-provided real-time threads (RT Threads). The programs used to measure the performance implement a producer/consumer model that one thread is a producer and another thread is a consumer. The benchmarks were performed on a Gateway2000 486DX2 66MHz system. Table 2 shows the performance of basic operations for comparison.

6 Summary

The goals of our user-level real-time threads are the dynamic management of thread attributes and the support of multiple thread models. We showed that the dynamic management of thread attributes can be achieved by realizing real-time threads at user-level. Introducing the user-level timer mechanism also makes the support of multiple thread models possible.

Our user-level real-time threads can also keeps compatibility with existing kernel-provided threads. They can coexist in the same environment, and existing applications still run without any modification.

The current real-time thread model is being implemented, and more accurate and various performance measurements will be completed.

Acknowledgments

We would like to thank members of Multimedia Platform Project for their various comments. We are also grateful to Prof. Tatsuo Nakajima and Mr. Takuro Kitayama for providing us with helpful information of RT-Mach.

References

[1] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the 13th Symposium on Operating System Principle*, October 1991.

[2] P. Barton-Davis, D. McNamee, R. Vaswani, and E.D. Lazowska. Adding Scheduler Activations to Mach 3.0. In *Proceedings of the USENIX Mach 3rd Symposium*, April 1993.

[3] E.C. Cooper and R.P. Draves. C Threads. Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, February 1988.

[4] R. Govindan and D.P. Anderson. Scheduling and IPC Mechanisms for Continuous Media. In *Proceedings of the 13th Symposium on Operating System Principle*, October 1991.

[5] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an Application Program. In *Proceedings of the Usenix Summer Conference*, June 1990.

[6] R.G. Herrtwich. The Role of Performance, Scheduling, and Resource Reservation in Multimedia System. In *Proceedings of International Workshop of Operating Systems of the 90s and Beyond*, Lecture Notes in Computer Science 563, Springer-Verlag, 1991.

[7] B.D. Marsh, M.L. Scott, T.J. LeBlanc, and E.P. Markatos. First-Class User-Level Threads. In *Proceedings of the 13th Symposium on Operating System Principle*, October 1991.

[8] S. Oikawa and H. Tokuda. User-Level Real-Time Threads: An Approach towards High Performance Multimedia Threads. In *Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, November 1993.

[9] S. Savage and H. Tokuda. RT-Mach Timers: Exporting Time to the User. In *Proceedings of the USENIX Mach 3rd Symposium*, April 1993.

[10] H. Tokuda and C.W. Mercer. ARTS: A Distributed Real-Time Kernel. *ACM Operating Systems Review*, Vol. 23, No. 3, 1989.

[11] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. In *Proceedings of USENIX Mach Workshop*, October 1990.

[12] H. Tokuda, Y. Tobe, S.T.-C. Chou, and J.M.F Moura. Continuous Media Communication with Dynamic QOS Control Using ARTS with an FDDI Network. In *Proceedings of ACM SIGCOMM'92*, August 1992.

[13] H. Tokuda and T. Kitayama. Dynamic QOS Control based on Real-Time Threads. In *Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, November 1993.

[14] H. Tokuda, S. Savage and C.W. Mercer. A Real-Time Thread Model for Continuous Media Applications. In Preparation.

[3] This version of RTC-Threads does not have real-time facilities yet.

Experience with a Prototype of the POSIX "Minimal Realtime System Profile"

T.P. Baker, Frank Mueller, Viresh Rustagi*
Department of Computer Science
Florida State University
Tallahassee, FL 32304-4019

Abstract

This paper describes experience prototyping the proposed IEEE standard "minimal realtime system profile", whose primary component is support for realtime threads. It provides some background, describes the implementation, and reports preliminary performance measurements.

1 Introduction

A *thread* is an independent sequential flow of control. Threads differ from processes by sharing a common virtual address space with other threads. Threads are widely accepted as a computational building block for both uniprocessor and multiprocessor environments. In uniprocessor environments, the thread model simplifies the programming of asynchronous operations. In multiprocessor environments, threads may also allow higher throughput, by utilizing more than one processor.

The idea of cheap concurrency or "lightweight processes" has been around in various forms for a long time, including support for coroutines in the Mesa programming language[13], and multitasking in the Ada programming language[18]. The Pthreads (POSIX Threads) proposal is intended to provide similar functionality for programs in the C language. It is based on considerable experience, including C-threads [2], Mach threads[16, 17], and Brown University threads [3]. Several commercial operating systems support multithreaded processes, including the Lynx[4], Sun[12, 14], and Chorus[1] operating systems.

The POSIX 1003.4a project[8] represents an attempt to achieve some degree of application portability for C programs, across operating systems that support threads. This is an extension of the POSIX application program interface, which generally follows the UNIX process model.

Threads are considered a "real time" extension to POSIX. IEEE draft standard P1003.13[9] proposes a set of realtime application profiles, i.e. subsets of the POSIX standard that are suitable for certain classes of realtime applications. Threads are a key feature of these profiles. In particular, the "Minimal Realtime System Profile" assumes a single process, with threads being the only form of concurrency within the system. The underlying hypothesis is that by not requiring support for the more complex POSIX features, the profile permits an implementation that will be satisfactory for realtime applications with very tight efficiency and timing predictability requirements.

The POSIX proposals are likely to have an impact on future realtime applications development, since they are being promoted as both U.S. Government and international (ISO/IEC) standards.

The draft Pthreads standard specifies the following services:

- *thread management:* initializing, creating, joining, and exiting threads.
- *synchronization:* mutual exclusion, and condition variables.
- *thread-specific data:* data maintained on per-thread basis.
- *thread priority scheduling:* priority management, preemptive priority scheduling, bounded priority inversion.
- *signals:* signal handlers, asynchronous wait, masking of signals, long jumps.
- *cancellation:* cleanup handlers, different interruptibility states.

2 Relationship to Ada Tasks

The Ada programming language[18] defines *tasks* as the only form of concurrent threads of control within a program. If the underlying operating system provides direct support for the POSIX (C-language) threads interface, it may be desirable to implement Ada tasks using this interface, by mapping Ada tasks to POSIX threads. Due to differences between the Ada and POSIX/C models, this mapping is not entirely straightforward.

The PART (POSIX Ada Real Time) project, at the Florida State University, is investigating the practicality of using POSIX threads to implement Ada tasks, especially in realtime applications. So far, a complete tasking implementation has been produced for the Ada 83 standard, using an implementation of P1003.4a Draft 6 layered over the Sun UNIX operating system[5]. Work is under way to extend this to the proposed new Ada 9X language standard [6, 19].

*This work was supported in part by the Ada Joint Program Office, via the U.S. Army CECOM HQ Software Engineering Directorate. However, the views reported here do not necessarily reflect a position of the sponsoring organization. Authors may be reached as "baker@cs.fsu.edu".

0-8186-5710-3/94 $3.00 © 1994 IEEE

3 A "Bare Machine" Implementation

To evaluate the suitability of a Pthreads-based Ada implementation for realtime applications, one must start out with a suitable realtime implementation of Pthreads. Experience with the FSU layered implementation, and other Ada tasking implementations, makes it clear that acceptable realtime performance is not achievable for an implementation layered over a conventional UNIX operating system. Efficiency is certainly an issue, but the main problem is predictability.

Most UNIX implementations impose unpredictable delays on user processes, due to preemptions by interrupt handlers and operating system processes. For an operating system to provide predictable timing of user threads, it must be designed with this objective in mind, from the hardware up. Some commercial realtime operating systems, such as LynxOS and Chorus, apparently have been designed in this way.

As a basis for performance testing of our Ada 9X implementation, we chose to port our existing layered implementation of Pthreads to a "bare" SPARCengine 1E[15]. We chose to do this rather than using an existing commercial realtime OS, for many reasons. Chief among these is that we needed source code, to tune the threads implementation to better support Ada (if necessary), and to take control over interrupts. We also were concerned that the commercial implementations of threads might be too full-blown to take advantage of the restrictions of the POSIX minimal profile, since they support multiple processes, file systems, and a variety of hardware devices. (Finally, there was concern that licensing restrictions would stand in the way of publication.)

The SPARCengine port of the FSU Pthreads library is intended to fit the POSIX minimal realtime systems profile. It runs on a "bare" machine, without any other operating system. It does not support multiple processes, and so operates in single virtual address space. It does not have a file system. At present the only devices supported are a serial port and a timer. These simplifications eliminate unpredictable time delays due to page faults, waiting for completion of I/O, and I/O completion interrupt processing.

The scope of this prototype implementation is limited to a subset of the proposed POSIX minimal realtime system profile. The criteria that governed the choice of this subset are:

- The implementation should be powerful enough to allow testing in a realtime context. This requires the following functionalities:
  - Dynamic creation and termination of threads
  - Synchronization primitives
  - A readable realtime clock
  - Timer support sufficient for periodic task scheduling
  - Output routines to print results
- The implementation should provide sufficient functionality to implement Ada tasking.

The design of the implementation is divisible into three main components:

1. Pthreads support. This implements the detailed functionality of the Pthreads standard, including the dynamic creation and termination of threads, the synchronization primitives, and thread scheduling. It is the largest component, but can be identical to a library implementation.
2. Machine-specific support. This includes code to save and restore register windows for context-switches, boot up the kernel, and provide time-keeping services. The boot code involves initializing memory mapping hardware and installing trap handlers. With the library implementation, all of these functions are performed by the underlying operating system. A bare-machine implementation must perform these functions for itself.
3. C language support, including basic I/O and memory allocation. These are functions provided by the standard C libraries, but the standard Sun Microsystems implementation of the C libraries makes calls to the operating system. Without the support of the operating system, these libraries need to be reimplemented.

The design of the Pthreads functionality was constrained to be *async safe*. A function is *async safe* if calling the function asynchronously will not cause any invariants to be violated, even if it is called from the handler of an interrupt that may be delivered at any time [7]. Even though POSIX does not require the Pthreads functions to be async safe, we chose to require it as a matter of quality. Async safety allows a user to build more responsive realtime systems. Furthermore, it is required to support Ada 9X.

A *single-threaded kernel* approach was used. Once a thread has entered the kernel, no other thread can enter the kernel until that thread has left it. The alternative, a *multi-threaded kernel*, where separate locks are associated with different kernel data structures, would allow more concurrency in a multiprocessor environment. For this to pay off, the cost of interprocessor locking must be low, relative to the time typically spent in kernel. Since we have only a single processor, the choice was clear; the overhead of fine-grained locking would result in poorer performance.

The source code of our bare-machine Pthreads kernel consists of approximately 3300 lines of C-code, of which approximately 1000 lines are new for the bare-machine version and the rest is reused from the library level implementation. The core image of the kernel is 49 kilobytes, as compared to 984 kilobytes for the full Sun UNIX kernel.

Reuse of most of the code from the layered FSU Pthreads library permits direct performance comparisons of the two implementations. Differences can be attributed to running on a bare machine, versus as a layer over the UNIX operating system.

This code was tested for both functionality and performance.

Functional Testing Functional testing was done using a set of 25 tests, derived from tests originally developed to test the layered version of the FSU Pthreads library. The features tested by these bare-machine tests

include:

- Thread management - creation, termination, join, detach
- Priority scheduling
- Mutexes - with and without priority ceilings
- Creating and destroying condition variables
- Timed conditional wait
- Thread specific data
- Setjmp/longjmp
- Signal handlers
- Cancellation and cleanup handlers

It was verified that the implementation could pass these tests, before performance testing began.

4 Absolute Performance Results

The performance tests were also derived from tests developed earlier for the layered version of the Pthreads library. These tests attempt to measure the specific performance metrics called out by Draft 6 of Pthreads.

Table 1 shows selected measurements of some of these metrics. The test programs use a dual loop timing analysis technique. The times reported are averages taken over 100,000 iterations. These measurements are compared to measurements taken earlier with the version of the Pthreads library layered over UNIX, on the same machine.

For the layered implementation, the time taken for 100,000 iterations of an operation ranges from about 100 milliseconds to 1 second. Though there was only one user process active, this is long enough that a system process might preempt, so the numbers shown here may be a bit high.

The metrics include:

- *Enter and exit Pthreads kernel.* This is the time taken to enter and immediately exit the kernel.
- *Mutex lock/unlock, no contention.* This is the time to perform a pair of mutex lock and unlock operations, under the assumption that a mutex is requested while unlocked.
- *mutex lock/unlock, contention.* This is the interval between an unlock by one thread and the return from a lock operation by another thread, which was suspended waiting for the mutex.
- *Semaphore synchronization.* This is one Dijkstra P operation plus one V operation. These are implemented on top of mutexes and condition variables.
- *Thread create, no context switch.* This measures the time taken to create a thread, excluding the context switch time.
- *setjmp/longjmp pair.* This is the time taken by a `setjmp` followed by a `longjmp`.
  The performance of a pair of `setjmp` and `longjmp` operations gives a lower bound on the overhead of a context switch, but a true context switch involves some additional overhead.
- *Thread context switch.* This is the time taken for a context switch.
- *Yield (1 thread).* This is the time taken by the *yield* operation when there is only one thread in the system.
- *Yield (2 threads).* In this case, there are two threads in the system.
- *Thread signal handler.* The measurements taken for signal handling reflect the time from sending a signal, by `pthread_kill`, till the signal is received.

Table 1: Performance of some Pthreads Operations

Pthreads Operations	Timings (μsecs)	
	Bare Machine	Layered over UNIX
enter and exit Pthreads kernel	1	1
mutex lock/unlock, no contention	3	3
mutex lock/unlock, contention	44	114
semaphore synchronization	60	103
thread create, no context switch	37	104
setjmp/longjmp pair	16	49
thread context switch	17	95
yield 1 operation	1	1
yield 2 operations	33	70
thread signal handler	55	92

5 Time Predictability Results

From the design of the implementation, we expect our bare-machine implementation to achieve predictable execution timing. The main cause of large deviations from the priority preemptive scheduling model has been eliminated, namely preemption of user threads due to scheduling of other processes, including operating system processes. The precision of timed wakeup events has also been improved, from ten down to one millisecond. With these improvements, we believe that the implementation of priority scheduling is strict enough that actual schedulable utilization will be very close to the theoretical predictions of schedulability analysis.

During debugging, we have already observed that the timing is remarkably consistent. This was evident in the reproducibility of failures due to race-condition problems between (earlier, incorrect versions of) the timer interrupt handler and the rest of the system.

We are currently working on benchmarks to measure the predictability of scheduling for actual task sets, to compare these against theoretical schedulability models, and to estimate the amount of overhead introduced by the Pthreads implementation.

The first test is based on a benchmark developed earlier for a preliminary design of a minesweeper trainer system for the U.S. Naval Coastal Systems Center. This consists of a set of six periodic threads, comprising a realtime simulation. Each thread has three phases. In the first phase, it reads the simulated state of other simulated subsystems from a global database. In the

second phase, it computes its own next state. In the third phase, it updates the global database. The read and update phases require locking the database, which is done via a single Pthread mutex. This is shown in pseudo-code in Figure 1. The thread periods, and the execution times of the three phases, are shown in Table 2.

```
for(;;) {
  pthread_mutex_lock(&shared_memory);
  input_data();
  pthread_mutex_unlock(&shared_memory);

  execute();

  pthread_mutex_lock(&shared_memory);
  output_data();
  pthread_mutex_unlock(&shared_memory);

  next_request[task] += period[task];
  if (next_request[task] >= simulation_time)
    break;

  /* suspend until next period */
  pthread_mutex_lock(&mutex[self]);
  do {
    pthread_cond_timedwait(&cond[self],
      &mutex[self], &next_request[self]);
    clock_gettime(CLOCK_REALTIME, &current_time);
  } while (next_request[task] > current_time);
}
```

Figure 1: Task Simulation Algorithm

Task	Period [ms]	Input [μs]	Execute [ms]	Output [μs]	Util.
1	62.5	2.0	44.80	2.0	71%
2	125.0	0.3	0.05	0.3	0%
3	166.7	1.6	27.80	1.6	16%
4	250.0	8.0	0.11	8.0	0%
5	500.0	3.2	5.02	3.2	1%
6	1,000.0	24.0	10.36	24.0	1%

Table 2: Task Set

Such a task system should be suitable for schedulability analysis, based on the Rate-Monotonic model. The objective of our benchmark is to determine how close the actual performance comes to this model.

In the benchmark, a bisection method is used to compute the *breakdown utilization*, at which the tasks can just barely be scheduled without missing any deadlines. This is done by varying a linear scaling factor, called *load factor*, which applies to the execution times of all phases of all the tasks.

The benchmark was run repeatedly over both UNIX and the bare-machine implementation with an initial target utilization of 90%. The results are shown in Figure 2.

It was observed that the timing of the benchmark

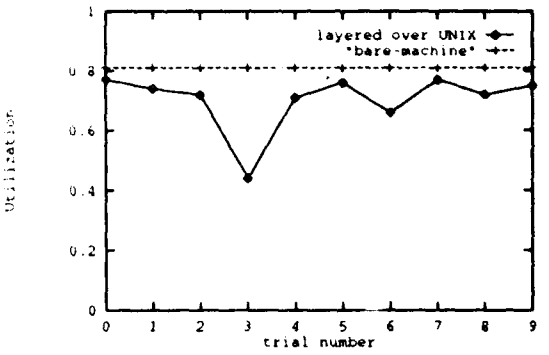


Figure 2: System Utilization for repeated Trails

over UNIX varies considerably at times. The bisection sometimes failed on its first iteration, thereby indicating that the breakdown utilization of 90% must reduced below 45%. The bisection would then proceed to terminate at a utilization around 44%. At other times, the bisection succeed for a trial of a certain load factor. Upon termination of the bisection, the same load factor was tried again but resulted in a failure. We adapted our algorithm to restart the bisection with the current load factor as the upper bounds upon these sporadic failures.

The bare-machine implementation produced very predictable results without any variation. The utilization of the benchmark was measured at 81%. The remaining 19% can be interpreted as the time consumed by the bare-machine implementation of Pthreads. Under UNIX, the benchmark utilization had its peak at 77% with a remaining 23% overhead due to the operating system and the layered Pthreads implementation. The smaller overhead of the bare-machine implementation can be attributed to the performance improvements discussed in the last section.

The occasionally large variations in the utilization under UNIX and the sporadic failures of the bisection algorithm seem to be due to operating system activities which occur at unpredictable times. These activities include process scheduling, CPU time accounting, and the processing of ethernet messages[1]. The unpredictability of the UNIX operating system limits its applicability for hard real-time systems. Hard real-time applications may not be able to safely achieve a high utilization under UNIX. A bare-machine implementation seems to permit a higher utilization for hard real-time applications, providing both predictability and an efficient use of the hardware.

6 Conclusions

We have implemented a sufficient subset of the Minimal Realtime System Profile to permit performance testing. The implementation supports preemptive priority scheduling, with a restricted form of priority ceiling emulation for mutexes. It supports a re-

[1]There was no local hard disk attached to the SPARCengine. The only asynchronous activities were due to clock and ethernet interrupts.

altime clock with microsecond precision, and timed events with millisecond precision, including the timeout for the wait operation on a condition variable. Experience with this implementation suggests that Pthreads can be implemented in a form that is suitable for realtime applications with hard timing constraints.

The absolute performance figures are encouraging. The performance of the bare-machine implementation is much better than that of the version layered over a full UNIX system. Part of this improvement is due to our algorithm for saving register windows to memory, which is different from that used by the commercial UNIX operating system. The other big contribution to the performance improvement is that our implementation avoids most of the overhead of UNIX system calls. User code executes in the same virtual address space as the kernel. This means kernel service calls can be ordinary subprogram calls, or even in-line macro calls, rather than traps. We also eliminate the overhead of demultiplexing service requests in the UNIX system call trap handler. This improvement seems specific to the minimal realtime systems profile. Running kernel and user processes in the same virtual address space would be unacceptable for a full POSIX implementation.

The experiments performed support the hypothesis that a bare-machine implementation can achieve excellent predictability. This provides the ability of this system to support *a priori* schedulability analysis, much in contrast to unpredictable systems such as UNIX.

Next, we plan to port the PART Ada runtime system implementation to the bare-processor Pthreads implementation, and test both the absolute speed and the timing predictability of Ada. This may require extending the functionality of the present implementation in some respects. Handlers need to be written for some traps that generate synchronous signals. For example, a mem_address_not_aligned trap should be processed to generate a SIGBUS for the current thread. The current C library support also needs some extensions.

Efforts will be made to flesh out the implementation in other respects, including support for timer-driven round-robin scheduling, and some debugging support.

References

[1] F. Armand, F. Herrmann, J. Lipkis, and M. Rozier, "Multi-threaded Processes in CHORUS/MIX", *Proceedings of EEUG Conference* (Spring 1990) 1-13.

[2] E. Cooper and R. Draves, "C threads". TR CMU-CS-88-154, Carnegie Mellon University, Dept. of CS (1988).

[3] T. Doeppner Jr., *A threads tutorial*, TR CS-87-06, Brown University, Dept. of CS (1987).

[4] Bill O. Gallmeister and Chris Lanier. "Early experience with POSIX 1003.4 and POSIX 1003.4a", *IEEE Symposium on Real-Time Systems*, IEEE Computer Society (1991) 190-198.

[5] E.W. Giering and T.P. Baker, "Using POSIX threads to implement Ada tasking: Description of work in progress", *TRI-Ada '92 Proceedings* (Nov 1992) 518-529.

[6] E.W. Giering, Frank Mueller, and T.P. Baker, "Implementing Ada 9x features using POSIX threads: Design issues", *TRI-Ada '93 Proceedings*, ACM (Sep 1993) 214-228.

[7] IEEE Portable Applications Standards Committee, *P1003.4a: Threads Extension for Portable Operating Systems (Draft 6)*, IEEE (Feb 1992).

[8] IEEE Portable Applications Standards Committee, *P1003.4a: Threads Extension for Portable Operating Systems (Draft 8)*, IEEE (Oct 1993).

[9] IEEE Portable Applications Standards Committee, *P1003.13: Information Technology - Standardized Applications Environment Profile - POSIX Realtime Application Support (AEP)* (Draft 5) (Feb 1992).

[10] Frank Mueller, "Implementing POSIX threads under UNIX: Description of work in progress", *Proceedings of the Second Software Engineering Research Forum* (Nov 1992) 253-261.

[11] Frank Mueller, "A library implementation of POSIX threads under UNIX", *Proceedings of the USENIX Conference* (Jan 1993) 29-41.

[12] M.L. Powell, S.R. Kleiman, S.Barton, D. Shah, D. Stein, and M. Weeks, "SunOS Multi-thread Architecture", *USENIX* (Winter 1991) 65-80.

[13] D. D. Redell et al., "Pilot: An operating system for a personal computer", *Communications of the ACM, Vol. 23, No. 2* (Feb 1980).

[14] D. Stein and D. Shah, "Implementing lightweight threads", *Proceedings of the USENIX Conference* (Summer 1992) 1-10.

[15] SUN Microsystems, Inc., *The SPARCengine 1E Card Family User's Manuals* Part No: 800-8137-02 (Apr 1990)

[16] A. Tevanian, R. F. Rashid, D. B Golub, D. L. Black, E. Cooper, and M. W. Young, "MACH threads and the UNIX kernel: The battle for control", *Proceedings of the USENIX Conference* (Summer 1987) 185-197.

[17] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao, "Real-Time MACH: towards a predictable real-time system", *USENIX MACH Workshop* (Oct 1990).

[18] U.S. Department of Defense. *Military Standard Ada Programming Language* ANSI/MIL-STD-1815A, Ada Joint Program Office (Jan 1983).

[19] Ada 9X Mapping/Revision Team, *Ada 9X Reference Manual: Draft Version 4.0*, Intermetrics, Inc., 733 Concord Avenue, Cambridge, Massachusetts 02138 *(available by anonymous FTP from ajpo.sei.cmu.edu)* (Sep 1993).

Availability of Source Code

The source code of the version of the Pthreads library layered over UNIX is available via anonymous ftp from `ftp.cs.fsu.edu` (128.186.121.27), in the file `/pub/PART/pthreads.tar.Z`. Other material (related publications) can be found in the same directory.

Session II: Scheduling I

Chair: Ted Baker
Florida State

An End-to-End Approach to Schedule Tasks with Shared Resources in Multiprocessor Systems

Jun Sun Riccardo Bettati Jane W.-S. Liu
Department of Computer Science
University of Illinois, Urbana-Champaign
Urbana, IL 61801

Abstract

In this paper we propose an end-to-end approach to scheduling tasks that share resources in a multiprocessor or distributed systems. In our approach, each task is mapped into a chain of subtasks, depending on its resource accesses. After each subtask is assigned a proper priority, its worst-case response time can be bounded. Consequently the worst-case response time of each task can be obtained and the schedulability of each task can be verified by comparing the worst-case response time with its relative deadline.

1 Introduction

Tasks in real-time systems often share resources, and semaphore-like operations are necessary to guarantee their mutual-exclusive access to critical sections. A previous study shows that careless use of semaphore operations can cause uncontrolled priority inversion, which occurs when a high-priority task is blocked by some low-priority tasks for an unpredictable amount of time [1]. We refer to the total length of time a task is delayed by lower-priority tasks due to resource contention as its *blocking time*. To ensure predictability, it is imperative to bound the blocking time of each task, as shown in [2]. Several effective solutions have been proposed for single processor systems; two well-known examples are the *Priority Ceiling Protocol* (PCP) [1] and the *Stack Based Protocol* (SBP) [3].

In multiprocessor and distributed systems concurrency and distribution complicate the resource contention problem. A task T_i can be blocked not only by a local task on the same processor due to local resource contentions, but also by a remote task that needs some global resources also needed by T_i. Rajkumar, et al. [4] extended PCP for single processor systems to multiprocessor systems and provided an initial solution for this problem. The extended protocol is known as the *Multiprocessor Priority Ceiling Protocol* (MPCP). According to MPCP, a resource needed by remote tasks on other processors is a *global resource*, and the processor on which a global resource resides is called its *synchronization processor*. When a task T_i gains access to a global resource, a *Global Critical Section* (GCS) server runs on the resource's synchronization processor on behalf of T_i. On each processor PCP is used to schedule both local tasks and GCS servers. Consequently, for each task, the total blocking time due to both local resource contention and global resource contention can be bounded, and whether each task can meet its deadline can be determined based on this blocking time by using the schedulability condition for the single-processor PCP.

However, the performance of MPCP is sometimes poor, especially for tasks on synchronization processors. One reason is that GCS servers on each synchronization processor always have higher priorities than local tasks. The priority inversion problem is reintroduced when a high-priority local task is delayed by GCS servers executing on behalf of lower-priority tasks.

In this paper we propose an end-to-end approach to scheduling tasks with shared resources and to analyzing their schedulability in multiprocessor systems. Section 2 gives an informal description of this approach and compares and contrasts it with MPCP. Section 3 presents in detail the procedure used in the end-to-end approach. Future work is discussed in section 4.

2 The End-to-End Scheduling Approach

From the viewpoint of end-to-end scheduling, a task that needs remote resources is viewed as a chain of subtasks in the following way. Each critical section

0-8186-5710-3/94 $3.00

associated with a remote resource is a subtask that executes on the synchronization processor of the remote resource. A segment that requires no resources or only local resources is also a subtask, and this subtask executes on the local processor. Subtasks of the same task collectively inherit the task's release time and deadline, and they execute in turn. Specifically, if task T_i has n subtasks, subtask $T_{i,1}$ is ready for execution at the release time of T_i, and subtask $T_{i,j}$ is ready for execution when subtask $T_{i,j-1}$ completes, for $j = 2, 3, \ldots, n$. The last subtask $T_{i,n}$ must complete by the deadline of T_i. If task T_i is a periodic task, this precedence relation holds for every instance of T_i.

The precedence relation among the subtasks of each task can be easily satisfied by using the phase-modification method proposed in [5]. Let $c_{i,j}$ be the worst-case response time of $T_{i,j}$. According to the phase-modification method, once we know $c_{i,k}$ for $k = 1, 2, \ldots, j-1$, we postpone the phase of the subtask $T_{i,j}$ by $\sum_{k=1}^{j-1} c_{i,k}$. This modification allows us to enforce the precedence relation between subtasks while treating the subtasks in each task as if there is no precedence relation between them. We will return to discuss how to bound the worst-case response times of subtasks on each processor using the schedulability condition in [5], provided that the subtasks are assigned fixed priorities and some single-processor synchronization protocol is used to control priority inversion. By summing up the worst-case response times of all its subtasks, we can determine the worst-case response time of each task, and therefore whether the task can meet its deadline.

Similar to MPCP, we allow nested resource accesses. However, we impose an additional restriction that all resources accessed in one nested critical section must reside on the same processor. In other words accesses to resources on different processors cannot be nested. One consequence of the end-to-end scheduling approach is that there is no need to control the accesses to remote, global resources differently from local resources. Each subtask that is a GCS server in MPCP model is local to its synchronization processor. All resource contentions are resolved locally and separately on each processor.

Table 1 gives an example, Example 1. In the table, T_i denotes a task; column *proc* lists the processor T_i is assigned to; ϕ_i is T_i's priority; p_i denotes T_i's period; and τ_i stands for T_i's processing time. The smaller the value of ϕ_i, the higher T_i's priority. The system in this example has two processors P_1 and P_2. There are two periodic tasks, T_1 and T_2, and one resource R. The deadline for each task is the end of its period. T_1 is assigned to P_1; T_2 and R are on P_2. The table lists the parameters of the tasks. Specifically, T_1 has three segments. The first and the last segments need no resource; they are executed on P_1, each with processing time 2. The middle segment requires the resource R; its processing time is 2. (The notation $t(R)$ in the *Segments* column indicates that the segment is a critical section that has duration t and accesses the resource R.) We note that the tasks can not be scheduled according to MPCP. Since T_1 needs to access R on P_2, there is a GCS server running on P_2 on behalf of T_1. This server has a higher priority than T_2. Since the processing time for this server is as long as T_2's period and T_2 will be blocked by the GCS server whenever the server executes, T_2 can not meet its deadline.

T_i	proc	ϕ_i	p_i	τ_i	Segments		
T_1	P_1	2	20	6	2	2(R)	2
T_2	P_2	1	2	1	1		

Table 1: Example 1 - A Simple System

In the end-to-end scheduling model, task T_1 is divided into three subtasks, $T_{1,1}$, $T_{1,2}$ and $T_{1,3}$. $T_{1,1}$ and $T_{1,3}$ execute on processor P_1 and need no resource, while $T_{1,2}$ executes on P_2 and needs resource R. $T_{1,1}$, $T_{1,2}$ and $T_{1,3}$ are dependent: the kth instance of $T_{1,1}$ (i.e., the instance of $T_{1,1}$ in its kth period) must complete before the kth instance of $T_{1,2}$ can begin execution. Similarly, the kth instance of $T_{1,3}$ cannot start execution until the kth instance of $T_{1,2}$ completes. Table 2 shows the parameters of the subtasks. $\tau_{i,j}$ is the processing time of subtask $T_{i,j}$, $f_{i,j}$ denotes the modified phase of $T_{i,j}$, and $\beta_{i,j}$ denotes the blocking time $T_{i,j}$ can experience.

$T_{i,j}$	proc	$\phi_{i,j}$	$p_{i,j}$	$\tau_{i,j}$	$\beta_{i,j}$	$c_{i,j}$	$f_{i,j}$
$T_{1,1}$	P_1	2	20	2	0	2	0
$T_{1,3}$	P_1	2	20	2	0	2	8
$T_{2,1}$	P_2	1	2	1	0	1	0
$T_{1,2}$	P_2	2	20	2(R)	0	6	2

Table 2: Example 1 - Using the End-to-End Approach to Schedule the Simple System

In this example, there is only one critical section, and therefore there is no blocking. The priorities of the subtasks are assigned on rate-monotonic basis. We see that the worst-case response time C_1 of the task T_1 is $c_{1,1} + c_{1,2} + c_{1,3} = 10$, which is less than 20, and the worst-case response time of T_2 is 1, and it is less than 2. We can therefore conclude that the deadlines of both tasks are always met.

Input :

1. Task set $\{T_i\}$. For each task T_i, the deadline D_i, period p_i, processing time τ_i, and resource accesses;
2. The task assignment mapping task set $\{T_i\}$ to processor set $\{P_k\}$;
3. The resource set $\{R_j\}$ and the resource assignment mapping $\{R_j\}$ to $\{P_k\}$.

Output : The conclusion whether the system can be scheduled and the priorities assigned to subtasks on each processor in the case the system is schedulable.

Step 1 : Map the given task set $\{T_i\}$ to a end-to-end task set $\{T_{i,j}\}$.

Step 2 : Assign priorities to subtasks.

Step 3 : Obtain the worst-case response time for each subtask.

Step 4 : Based on the results obtained in Step 3, analyze the schedulability for the whole system.

Figure 1: Pseudo-Code of the End-to-End Scheduling Procedure

3 Schedulability Analysis

We now describe how to choose the priorities for subtasks and determine their worst-case response times. We confine our attention to the case where tasks are periodic and their subtasks are assigned fixed priorities. However, the subtasks of each task may be assigned different priorities.

Figure 1 gives the pseudo-code description of the end-to-end scheduling procedure.

Step 1 : Map the given task set to an end-to-end task set

Following the rules below, Step 1 breaks up each task T_i in the given task set into a chain of n_i subtasks $T_{i,j}$ in the corresponding end-to-end task set :

1. Each subtask $T_{i,j}$ is either a critical section that requires some remote resources or a segment that requires no resource or only local resources. If a task has nested resource accesses, each outermost critical section is mapped to a subtask.
2. A subtask that requires no resource or only local resources is on the local processor of T_i. A subtask that requires remote resources is on the synchronization processor of the remote resources.
3. For every $j = 1, 2, \ldots, n_i - 1$, consecutive subtasks $T_{i,j}$ and $T_{i,j+1}$ are on different processors.

Rule 3 is not necessary for the correctness of the later discussion. However it allows us to obtain a tighter upper bound for the response time of each subtask.

Example 2 illustrates the rules described above. In this example there are four resources and three processors. Resource R_1 is assigned to processor P_1; R_2 and R_3 to P_2; and R_4 to P_3. Task T_1 is a periodic task. It has 10 segments, as shown by Figure 2. The shaded segments denote that T_1 requires some resources during those time intervals.

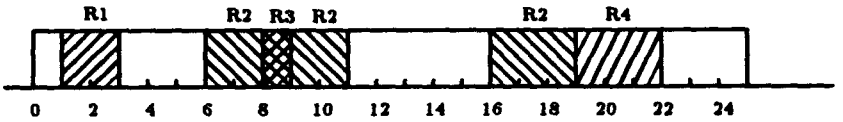
According to Step 1, T_1 is mapped into 6 subtasks, as shown by Table 3. The segment from time 0 to time 6, denoted as (0,6], is mapped onto one subtask $T_{1,1}$ because during this time interval, T_1 either does not require any resources or only requires local resources. According to rule 3, we map it onto one subtask, and it runs on the local processor, P_1. Similarly, segment (6,10] is mapped onto the subtask $T_{1,2}$ because the accesses to R_2 and R_3 are nested and only the outmost critical section becomes a subtask. This subtask runs on processor P_2. Segments (16,19] and (19,22] are two different subtasks, $T_{1,4}$ and $T_{1,5}$, because they access different remote resources. They run on P_2 and P_3 respectively. The segments (10,16] and (22,24] are mapped onto $T_{1,3}$ and $T_{1,6}$. They are both on P_1.

T_i	proc	p_i	$\tau_{i,j}$	Segment		
$T_{1,1}$	P_1	50	6	1	$2(R_1)$	3
$T_{1,2}$	P_2	50	5	$2(R_2)$	$1(R_2, R_3)$	$2(R_2)$
$T_{1,3}$	P_1	50	5	5		
$T_{1,4}$	P_2	50	3	$3(R_2)$		
$T_{1,5}$	P_3	50	3	$3(R_4)$		
$T_{1,6}$	P_1	50	3	3		

Table 3: Example 2 - Subtasks Assignment

Step 2 : Assign priorities to subtasks

Several methods can be used to assign priorities. Rate-monotonic assignment is a possible choice. Other choices include :



T_i	proc	p_i	τ_1	Segments								
T_1	P_1	50	25	1	$2(R_1)$	3	$2(R_2)$	$1(R_2, R_3)$	$2(R_2)$	5	$3(R_2)$	$3(R_4)$	3

Figure 2: Example 2 - Task T_1

- Global-deadline-monotonic assignment: the priority of a subtask is based on the global relative deadline, D_i, the deadline of the task T_i; the shorter D_i is, the higher priority $T_{i,j}$ has.
- Effective-deadline-monotonic assignment: the priority of a subtask $T_{i,j}$ is chosen based on subtask's effective relative deadline. The effective relative deadline $ED_{i,j}$ of $T_{i,j}$ in a task T_i with n_i subtasks is:

$$D_i - \sum_{k=j+1}^{n_i} \tau_{i,k}$$

$T_{i,j}$ must complete at $ED_{i,j}$ units of time after T_i is released in order for T_i as a whole to complete in time.

Table 4 lists the priorities of subtasks in Example 3 with their priorities assigned based on their effective relative deadlines.

T_i	proc	ϕ_i	p_i	$\tau_{i,j}$
$T_{1,1}$	P_1	31	50	6
$T_{1,2}$	P_2	36	50	5
$T_{1,3}$	P_1	41	50	5
$T_{1,4}$	P_2	44	50	3
$T_{1,5}$	P_3	47	50	3
$T_{1,6}$	P_1	50	50	3

Table 4: Example 2 - Priority Assignment Based on Subtasks' Effective Deadlines

Step 3 : Determine the worst-case response times for subtasks

After Step 2 we have a set of subtasks on each processor, in which (1) every subtask requires either no resource or local resources and (2) every subtask has a fixed priority. Resource-access-control protocols for single-processor systems can be used to prevent deadlocks and uncontrolled priority inversion. Both PCP and SBP can be used in this case. Furthermore, we can obtain the worst-case blocking time $\beta_{i,j}$ for each subtask $T_{i,j}$. Consequently the worst-case response time $c_{i,j}$ for each subtask can be computed according to the following equation. The derivation for this equation can be found in [5].

$$c_{i,j} = \frac{\sum_{T_{k,l} \in H_{i,j}} \tau_{k,l} + \beta_{i,j}}{1 - \sum_{T_{k,l} \in H'_{i,j}} u_{k,l}} \quad (1)$$

In this equation $H_{i,j}$ is the set of subtasks that (1) are on the same processor as $T_{i,j}$, (2) are of different tasks than T_i, and (3) have priorities equal to or higher than $T_{i,j}$. $H'_{i,j}$ is a subset of $H_{i,j}$ in which every subtask has a higher priority than $T_{i,j}$. $u_{i,j}$ is the processor utilization factor of $T_{i,j}$. Again, $\beta_{i,j}$ is the maximum blocking time $T_{i,j}$ can experience. For both PCP and SBP, $\beta_{i,j}$ can be approximated by $MAX(S_{k,l})$, where $S_{k,l}$ is the maximum duration of critical sections for all possible $T_{k,l}$ that (1) is on the same processor as $T_{i,j}$ and (2) has lower priorities than $T_{i,j}$.

Step 4 : Check schedulability for the whole system

From the results obtained in previous step, the worst-case response time for T_i can be obtained by summing up all response times of its subtasks :

$$C_i = \sum_j c_{i,j} \quad (2)$$

If $C_i > D_i$, where D_i is the relative deadline of task T_i, we report failure for this task set. If all tasks pass this test, we report success.

4 Conclusions

In the previous section we present a procedure for applying the end-to-end approach to scheduling tasks with shared resources in a multiprocessor system and

analyzing the schedulability. In order to make this approach practical, some formulas need to be improved and problems which may arise in practice need to be addressed. For example, the upper bound for worst-case response time given by Eq. (1) sometimes is not satisfactory, especially for subtasks with low priorities. A method based on time-demand analysis has been developed to give a much tighter bound and will be presented in a future paper.

Another practical problem arises when we fix the subtasks' phases to enforce the execution precedence among them. In order to make the modified phases consistent and meaningful in a multiprocessor or distributed system, clocks on all processors have to be strictly synchronized, which can be difficult to achieve in practice. We can allow some clock drift among processors, provided that the drift is within a maximum limit of δ time units. Extra δ time units can be added to the worst-case response time for each subtask obtained in the previous section, and the execution precedence relations among subtasks will be safely enforced.

Another solution to this problem is to use dynamic phasing for subtasks instead of static phasing used in this paper. In other words, a subtask can be triggered to start as soon as its previous subtask finishes. We are currently working on the schedulability analysis for such systems.

An alternative way to map tasks to subtasks is to map all critical sections, both for local resources and for remote resources, into subtasks. The resultant task system has end-to-end processing not only across processors but also within each processor. A study in [6] has shown that schedulability analysis for end-to-end processing within a processor is possible and promising. We are currently studying the schedulability analysis for such systems.

In this paper we assume that all resources accessed in one nested critical section must be on the same processor. This assumption in general can be overly restrictive. We will address this problem from the point of view of both resource access control and task/resource assignment. Ideally we want to assign resources to processors to minimize the number of nested critical sections that access resources on more than one processor.

In many ways, the end-to-end scheduling approach can be viewed as a divide-and-conquer approach: it divides the problem by mapping the given task set onto an end-to-end task set where each processor becomes relatively independent. It then resolves the local resource contention on each processor. Finally combines the results to obtain a global solution. This merit leads to a reduction in the complexity of the resource contention problem.

Acknowledgements

This work was partially supported by NSF contract No. NSF MIP 92-22408 and US Navy ONR contract No. N0001492J1815. The authors thank all members in Real-Time Systems Laboratory at University of Illinois for many informative and inspiring discussions, in particular, Too-Seng Tia for many in-depth discussions about MPCP model.

References

[1] L. Sha, R. Rajkumar and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization". *IEEE Transactions on Computers*, Vol. 39, No. 9, September 1990.

[2] R. Rajkumar, *Task Synchronization In Real-Time Systems*, Kluwer Academic Publishers, Boston 1991.

[3] T. P. Baker, "A Stack-Based Resource Allocation Policy for Real-Time Processes". *Proceeding of the 11th Real-Time Systems Symposium*, pp. 191-200, 1990.

[4] R. Rajkumar, L. Sha and J. P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors". *Proceeding: Real-Time Systems Symposium*, pp. 259-269, 1988.

[5] R. Bettati, "End-to-End Scheduling to Meet Deadlines in Distributed Systems". *Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign*, March 1994.

[6] M. G. Harbour, M. H. Klein, and J. P. Lehoczky, "Timing Analysis for Fixed-Priority Scheduling of Hard Real-Time Systems", *IEEE Transactions on Software Engineering*, Vol. 20, No. 1, pp. 13 - 28, January 1994.

Appropriate Mechanisms for the Support of Optional Processing in Hard Real-Time Systems*

N. C. Audsley, R. I. Davis, A. Burns and A. J. Wellings.

Real-Time Systems Research Group, Dept. of Computer Science, University of York, UK.

Abstract

It has been recognised that future hard real-time systems need to be more flexible than current scheduling theory permits. One method of increasing flexibility is the incorporation, at run-time, of optional components into processes with hard deadlines. Such components are not guaranteed offline, but may be guaranteed at run-time if sufficient resources are available. This is achieved by providing mechanisms within the kernel for run-time monitoring of spare processor capacity and its subsequent assignment to requesting processes. This paper examines these mechanisms within the context of fixed priority pre-emptive scheduling.

1. Introduction

The next generation of hard real-time systems need to be flexible, adaptive and able to exhibit intelligence. This is contrary to the relatively inflexible and static approaches enforced by scheduling theory and kernel design today: both must advance before such flexibility can be realised in applications.

One method of achieving improved flexibility is the provision of optional components, not afforded offline guarantees, but executed at run-time if sufficient resources are available. Classically, such components are embodied in soft real-time processes, executed when no guaranteed (i.e. hard) process is runnable. It is our contention that greater flexibility and utility is obtained by permitting critical processes to request time for optional components [4]. Such components may be bounded and need to complete once started or be guaranteed a minimum time (unbounded or bounded). Under these circumstances it becomes apparent that we require the ability to guarantee, at run-time, execution time for optional components.

The focus of this paper is upon kernel mechanisms to provide guaranteed execution time for optional components of hard processes within fixed priority pre-emptive systems.

Assuming that in safety-critical systems on-line guarantees of execution time for optional components cannot be provided at the expense of reduced predictability [4], a two-tier view of scheduling is taken:

- initially, offline guarantees are given to hard processes, providing a guaranteed minimum service;
- then, on-line guaranteed execution time is provided for additional components at run-time without violating the guarantees made offline.

The first tier is provided by fixed priority scheduling (rapidly becoming a *de facto* standard in hard real-time systems): scheduling theory is now available for the sufficient analysis of complex process sets [2]. The second tier utilises the inherent spare processor capacity at run-time to provide additional, guaranteed execution time, to requesting hard processes. Within this approach, the following issues become apparent:

- appropriate programming models to express additional components;
- identification of spare capacity at run-time;
- assignment of spare capacity to requesting processes.

Many programming models have been proposed, e.g. the Imprecise Model of Liu *et al* [10] and the Unbounded Model of Audsley *et al* [3]. However, the discussion of such models is beyond the scope of this paper. The rest of this paper discusses kernel mechanisms to support the identification and assignment of spare capacity at run-time to requesting hard processes within fixed priority pre-emptive systems.

We assume that the kernel, as opposed to application processes, is responsible for the identification and assignment of spare capacity at run-time. This is in-keeping with Rushby's view that a kernel for a safety-critical system should contain any mechanisms whose use by one application process may affect another application process which is crucial to the overall operation of the system [11]. In contrast, it is assumed that the demand for spare capacity is application process oriented. Amongst competing processes, a kernel policy decides, at run-time, to which process spare capacity is assigned.

The remainder of this introduction details our terminology. Section 2 describes mechanisms for on-line detection of spare capacity. Kernel level representation

* This work is supported, in part, by the UK Science and Engineering Research Council, grant number GR/H 39611.

0-8186-5710-3/94 $3.00 © 1994 IEEE

and management of detected spare capacity are discussed in section 3. Implementation issues are discussed in section 4, with section 5 offering our conclusions.

1.1 Terminology

Process τ_i has (unique) fixed priority i, worst-case execution time (WCET) C_i, deadline D_i, and is invoked at periods defined by T_i (for sporadic processes T_i is the minimum inter-arrival time). The set of processes with higher priorities than τ_i is given by $hp(i)$, the set of processes with equal or lower priority than τ_i is given by $lp(i)$. The set of higher priority levels than i is given by $hpl(i)$, with the set of priority levels equal to or lower than i is given by $lpl(i)$.

2. Identification Of Spare Capacity

Given the need to provide 100% deadline predictability for hard processes, it is inevitable that the processor and other resources will be under-utilised at run-time. This occurs for many reasons [3], including pessimistic WCET analysis, hardware speed-ups (e.g. cache, pipeline) etc. We term the resources not required at run-time as *spare capacity*. Two forms of spare capacity may be identified:

> ***Gain Time*** - processor time guaranteed to a crucial process offline but not required at run-time.
>
> ***Slack Time*** - processor time not utilised by guaranteed executions at run-time.

Since gain time has been guaranteed offline as part of a crucial process's WCET, if it is assigned to another process at run-time, the latter inherits the guarantee afforded to the original process. Section 2.1 discusses a mechanism for the detection of gain time. In general, a guarantee cannot be afforded to a process assigned slack time (since it was not necessarily guaranteed to a process offline). However, mechanisms exist which relax this restriction. One example is discussed in section 2.2.

2.1 Gain Points: A Mechanism for the Detection of Gain Time

Several approaches have been proposed which facilitate the detection of gain time. Haban *et al* place software triggers at the end of basic blocks in process code to measure actual execution time [8]. This is then compared with the pre-determined WCET of the block to calculate gain time. In a similar way, Dix *et al* allow the insertion of *milestones* into process code [7]. When the milestone is reached the maximum remaining execution time of a process is communicated to the kernel. The motivation of the milestone is to declare the point in its computation where it has completed its major computation (i.e. only has housekeeping functions remaining), although it could be used to detect gain time: milestones inserted at the end of basic blocks would detect gain time as in Haban's approach.

The property of both these approaches is that gain time is detected *after* it has been generated, at a relatively coarse granularity. In general, we wish to be aware of spare capacity as early as possible (implying a finer detection granularity than Haban's or Dix's approach): the sooner it can be determined, the sooner it can be usefully utilised.

Consider the following program fragment:

```
IF condition THEN 16 units ELSE 4 units FI
```

If the condition holds, the WCET of the statement is 16 units, otherwise 4 units. However, WCET analysis will have calculated that the WCET of the statement is 16 units. Hence, if the `else` clause is executed, 12 units of gain time will become apparent. The earliest place that this gain time can be detected is after the condition has been determined, prior to execution of the `else` clause. Here, a *gain point* is inserted into the code. This takes the form of a software trigger, informing the kernel of the gain point value via a system call (to the kernel). Where the gain point has a fixed value it is termed *static*.

This approach can be extended for other control-flow language constructs. For example, in languages suitable for hard real-time systems a maximum loop count is declared at compilation time. When the actual number of iterations is known (either prior to loop entry or on loop exit) a *dynamic* gain point can be declared whose value is equal to the number of iterations not required, multiplied by the WCET of the loop body.

We note that gain time, in general, is detected sooner using gain points than in either Haban or Dix's approach. Further uses of gain points, for other common language structures, are given in [1]. The gain point approach is applicable to both sporadic and periodic processes.

Gain points (i.e. code for calculating the value and making the kernel call) are inserted automatically during compilation or WCET analysis. However, it remains possible that a rogue process could report more gain time than detected (possibly causing failure if that gain time is guaranteed to a requesting process). To detect this, when a gain time call is made the kernel evaluates the condition:

$$C_i \geq g_i + A_i + c_i^{rem} + v \qquad (1)$$

where g_i is the gain time detected by τ_i so far in its current execution; A_i is the actual execution time used so far by the current execution; c_i^{rem} is the WCET of the process code from the gain point to completion (calculated during WCET analysis and parameterised within loops); v is the amount of gain time reported by the call. If condition (1) does not hold, the amount of gain time reported by the process is erroneous.

Note that $g_i + A_i$ is the WCET of the process up to the gain point and $c_i^{rem} + v$ the remaining WCET after the gain point.

This approach can be extended to permit the effects of pipelines and other processor accelerators to be monitored. Assuming that the condition (1) holds, the amount of gain time due to these hardware features is given by:

$$e = C_i - \left(g_i + A_i + c_i^{rem} + v\right)$$

Thus, the actual gain time recorded by the kernel is $e+v$.

Implementation is discussed in section 4.

2.2 The Approximate Slack Stealer: A Mechanism for the Detection of Slack Time

Assuming that gain time is detected on-line via gain points, the only remaining spare capacity is slack time, which occurs dynamically at run-time if the utilisation of the system is less than 100%. Conventionally, slack time cannot be guaranteed to processes. However, recent research has shown that a proportion of the available slack time can be guaranteed to requesting processes at run-time. Essentially, at any time, the amount of processor time that can be stolen immediately from hard processes is calculated, whilst maintaining the offline guarantees given to those processes: the execution of hard processes is postponed, allowing other non-guaranteed executions to occur. The amount of processor time that can be stolen is termed the *slack*.

The *Optimal Slack Stealer* algorithm relies on a pre-computed table to define the slack present at each invocation of each hard process [9]. The method suffers two drawbacks: only periodic processes were considered; the size of the table is, in general, non-polynomial. These problems were addressed by the *Dynamic Slack Stealer* [5], which enables the amount of slack available to be calculated on-line: at time t, the amount of slack at priority level i is equal to the amount of time not required by processes $hp(i) \cup \tau_i$ before the next deadline of τ_i. The amount of slack that be guaranteed to a process at priority level i is the minimum slack at priority levels $lpl(i)$ [5]. This approach has non-polynomial complexity - inappropriate for use within the kernel.

Based upon the Dynamic Slack Stealer, a class of on-line Approximate Slack Stealing algorithms can be derived. These have been shown to be more effective than bandwidth preserving algorithms [6].

At any point in time, Approximate Slack Stealing algorithms provide a lower bound on the slack time available (i.e. for a given interval, the slack detected is less than or equal to the exact amount of slack). We now provide the derivation of one such algorithm. The following are assumed available from the kernel at time t:

$x_i(t)$ - the earliest possible next release of τ_i ;

$d_i(t)$ - the next deadline of an invocation of τ_i (if the current invocation is complete then $d_i(t) = x_i(t) + D_i$);

$c_i(t)$ - the remaining (worst-case) execution time of the current invocation of τ_i.

Note that $x_i(t)$ and $d_i(t)$ are measured relative to time t.

The exact amount of slack time available at priority level i in $[t, t+d_i(t))$ whilst guaranteeing τ_i meets its deadline can be found by viewing the interval as comprising a number of level i busy and idle periods (i.e. periods where processes of priority i or higher are executing or not, respectively). Any level i idle time in the interval can be swapped for τ_i computation without causing the deadline to be missed. A lower bound on this level i idle time is found by obtaining an upper bound on the time processes $\tau_j \in hp(i) \cup \tau_i$ require in $[t, t+d_i(t))$. The maximum computation that τ_j performs in the interval is given by[1]:

$$I_j(t, d_i(t)) = c_j(t) + f_j(t, d_i(t))C_j + \min\left(C_j, \left(d_i(t) - x_j(t) - f_j(t, d_i(t))T_j\right)_0\right)$$

Where $f_j(t, d_i(t))$ is the number of complete invocations of τ_j in $[t, t+d_i(t))$:

$$f_j(t, d_i(t)) = \left\lfloor \left(d_i(t) - x_j(t)\right) / T_j \right\rfloor_0$$

Thus, $I_j(t, d_i(t))$ comprises three components: τ_j computation outstanding at t; $f_j(t, d_i(t))$ complete invocations of τ_j and a partially complete final invocation. A lower bound on the level i slack at time t, $S_i(t)$, is given by:

$$S_i(t) = \left(d_i(t) - \sum_{\forall j \in hp(i) \cup \tau_i} I_j(t, d_i(t)) \right)_0$$

To enable the assignment of available slack time to requesting processes, we must ensure that all hard processes still meet their deadlines. Hence, the maximum amount of slack that can be guaranteed at priority level i (i.e. assigned to a requesting process) is given by:

$$S_i^{max}(t) = \min_{\forall j \in lp(i)} S_j(t)$$

The approach is applicable to periodic and sporadic processes. In general, the complexity of the approach is $O(n)$ for determining the slack at one priority level and $O(n^2+n)$ for determining the slack at all levels. It is noted that the latter is bounded, therefore appropriate for use by the kernel. Approximate Slack Stealing algorithms have been explored further by Davis [6]. Implementation issues are discussed in section 4.

[1] $(x)_0$ represents $\max(x,0)$: its minimum value is thus 0.

3. Management Of Spare Capacity

In this section show how the spare capacity detected by the two mechanisms can be integrated into a joint framework to allow efficient assignment to requesting processes.

3.1 Representation of Spare Capacity

We assume that the values $x_i(t)$, $d_i(t)$, A_i, $S_i(t)$ and g_i are held by the kernel, together with, g_i^*, the amount of gain time assigned from priority level i during the current execution of τ_i. Thus, the storage requirements are $O(n)$ in the number of processes.

3.2 Management of Spare Capacity Tuples

Since gain time has been guaranteed at a particular priority level by offline feasibility analysis, in general, it must be utilised in preference to the normal execution of a lower priority process. Otherwise, one of the fundamental assumptions of fixed priority feasibility analysis, that processes execute as soon as possible, is violated with the possibility that guaranteed deadlines may be missed. Also, values of $S_i(t)$ must be updated as time progresses: if τ_i executes, the amount of slack available at $hpl(i)$ must be decreased by the amount of execution of by τ_i.

The underlying management issue is of correctly ageing spare capacity. This can be achieved by modifying the fundamental principle contained in the Priority Exchange algorithm [12]: if gain time exists at priority level i when τ_i completes, the gain time at priority level i is moved to priority level i-1.

Whilst updating of available gain time and slack time could occur every time unit, the overhead would become large (without hardware support). Alternatively, updates could occur whenever a demand for spare capacity is made. However, this would require that the kernel record the executing processes and the amount of time they executed since the last update. This is complex since, potentially, many invocations of the guaranteed processes may occur. The approach adopted in this paper is to update the spare capacity at a context switch or request for spare capacity (other approaches are given in [1]).

At an update, let the executing process be τ_i, with the elapsed time since the previous update be E (τ_i will be the only executing process in this interval). To update the available spare capacity, the kernel must:

(1) decrease the available slack at $hpl(i)$ by E;

(2) if τ_i has completed, increase the slack at levels $lpl(i)$ by $g_i - g_i^*$ and set $g_i=0$ and $g_i^*=0$.

Note, if the processor was idle in the interval since the last update, the slack at priority levels $lpl(1)$ is decreased by E.

By using the above rules, the guarantees associated with gain time and slack time are maintained, without causing deadline failure of other processes.

Note that when the Approximate Slack Stealing algorithm is executed, the value of $c_i(t)$ is required. The management approach adopted above implies that since gain time is held separately to slack time, the value of $c_i(t)$ used when calculating slack is given by:

$$c_i(t) = C_i - A_i$$

This management approach increases context switch time by a bounded amount, $O(n)$ in the number of hard processes.

3.3 Assigning Spare Capacity

Given the management of spare capacity as described above, the assignment of spare capacity to requesting processes becomes relatively straightforward. Consider hard process τ_i requesting additional computation time at time t before its deadline at $t+d_i(t)$. A number of sources of guaranteed spare capacity may be checked, including:

(1) check gain time at priority level i (if $g_i > g_i^*$ then gain time is available for assignment);

(2) check gain time at priority levels $hpl(i)$;

(3) check gain time at priority levels $j \in lpl(i)$ where $t+d_j(t) \le t+d_i(t)$;

(4) check slack time at priority levels $lpl(i)$ (i.e. evaluate $S_i^{max}(t)$);

(5) if the time at which the slack at priority levels $lpl(i)$ was last calculated is prior to t, we may check for the possibility of additional slack by recalculating the slack for priority levels $lpl(i)$, then repeating (4).

Clearly, more than one of the above can be used in the provision of spare capacity to a single request.

The complexity of (1) is O(1). In general, the complexity of (2), (3) and (4) are $O(n)$, with (5) being $O(n^2+n)$. A fast decision can be made using (1), with greater cost incurred by the other approaches, although a greater amount of gain time may be found with an associated increased chance of being able to honour the request for spare capacity.

An $O(n)$ approach for guaranteeing an amount of computation before an arbitrary deadline (i.e. not necessarily the deadline of the requesting process) is given in [14].

After spare capacity has been assigned, adjustments to the available spare capacity must be made. If gain time at priority level i is assigned, g_i^* is increased; if slack time at priority level i is assigned, the available slack at $lpl(i)$ is decreased.

4. Implementation Issues

It has been argued that the requirement for small and bounded kernel overheads precludes the use of complex on-line scheduling techniques [13]. It is the authors' contention that to provide increased flexibility via re-use

of spare capacity requires relatively complex, but bounded, on-line kernel algorithms to be employed. The reliability of the kernel should not be affected if it is constructed using the same engineering procedures as employed during the application software life-cycle. The fundamental issue is not one of on-line complexity, but of being able to guarantee the feasibility of processes offline. Hence, the use of on-line scheduling must minimise any effects it has upon process feasibility.

The number of gain points inserted into process code can be high. Ideally, their insertion should not affect the feasibility of a process by increasing its WCET. This is not always possible unless some gain points are omitted. However, overheads can be reduced by only inserting gain points whose value will be at least the cost of the gain time kernel call. In this case, some gain time will be detected late, although will show up as an efficiency speed up next time a gain point kernel call is made. Now, assuming that all processes are feasible, any gain points not originally inserted could now be placed into process code whilst the system remains feasible. Alternatively, gain time could be reported at a context switch or request for spare capacity (i.e. at an update - see section 3.2) [1].

The execution of the Approximate Slack Stealing algorithm need only occur when insufficient spare capacity is available for outstanding requests. Then, the algorithm is only executed if sufficient spare capacity exists to execute the algorithm itself. Thus, the feasibility of processes is not affected by executions of the algorithm. We note that the algorithm can also execute instead of idling the processor.

Whilst the above reduces the effects of spare capacity detection and assignment upon the feasibility of processes, some other overheads must be taken into account, e.g. spare capacity updates. The approach of section 3.3 implies that additional information must be collected and stored at run-time. Also, the manipulation of the run queue may become more complex when a process is assigned spare capacity at a different priority level to its own: the effects of this can be reduced by, for example, restricting a process to spare capacity at one priority level other than its own.

5. Conclusions

This paper has illustrated how optional components, not guaranteed offline, can be guaranteed computation time at run-time (if sufficient spare capacity is available). Spare capacity is detected by the Gain Point mechanism and the Approximate Slack Stealing algorithm. The management of detected spare capacity has been described, enabling its efficient assignment to requesting processes at run-time. The approach described can be extended for more flexible process characteristics [1,5,6], e.g. resource sharing (i.e. process blocking), precedence constrained processes; and process release jitter.

Currently, the mechanisms described within this paper are being incorporated into the DrTEE hard real-time kernel. The initial results indicate that the feasibility of processes is not unduly affected by using these mechanisms, with the ability to re-use spare capacity at run-time adding to the amount of computation time available to hard processes at run-time, so improving the utility and flexibility of the resultant system.

References

[1] Audsley, N. C. "Flexible Scheduling of Hard Real-Time Systems". Dept. of Comp. Sci., Univ. of York, UK. D.Phil. Thesis. (1993).

[2] Audsley, N. C., Burns, A., Richardson, M. F., Tindell, K. W. and Wellings, A. J. "Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling". *Soft. Eng. J.* 8, (5): 284-292. (1993).

[3] Audsley, N. C., Burns, A., Richardson, M. F. and Wellings, A. J. "Incorporating Unbounded Algorithms Into Predictable Real-Time Systems". *Comp. Sys. Sci. and Eng.* 8, (3): 80-89. (1993).

[4] Burns, A. and Wellings, A. "Criticality and Utility in the Next Generation". *Real-Time Sys.* 3, (4): 351-354. (1991).

[5] Davis, R. I., Tindell, K. and Burns, A. 1993. "Scheduling Slack Time in Fixed Priority Pre-emptive Systems". *Proc. IEEE Real-Time Sys. Symp.*, pp. 222-231. (1993).

[6] Davis, R. I. "Approximate Slack Stealing Algorithms for Fixed Priority Pre-emptive Systems". Dept. of Comp. Sci., Univ. of York, UK. YCS 217. (1993).

[7] Dix, A., Stone, R. F. and Zedan, H. S. M. "Design Issues for Reliable Time-Critical Systems", *Proc. of Workshop on Real-Time Sys.*, York, UK. (1989).

[8] Haban, D. and Shin, K. G. "Application of Real-Time Monitoring to Scheduling Tasks With Random Execution Times". *IEEE Trans. on Soft. Eng.* 16, (12). (1990).

[9] Lehoczky, J. P. and Ramos-Thuel, S. "An Optimal Algorithm for Scheduling Soft Aperiodic Tasks in Fixed Priority Pre-emptive Systems", *Proc. IEEE Real-Time Sys. Symp.*, pp. 110-123. (1992).

[10] Liu, J. W. S., Lin, K. J., Shih, W. K., Yu, C. S., Chung, J. Y. and Zhao, W. "Algorithms for Scheduling Imprecise Computations". *IEEE Comp.*, May: 58-68. (1991).

[11] Rushby, J. "Kernels for Safety?" pp. 310-320 in *Safe and Secure Computing Systems*, Anderson, E. T. (ed). (1987).

[12] Sprunt, B., Lehoczky, J. P. and Sha, L. "Exploiting Unused Periodic Time For Aperiodic Service Using the Extended Priority Exchange Algorithm". *Proc.IEEE Real-Time Sys. Symp.*, pp. 251-258. (1988)

[13] Stankovic, J. A. and Ramamritham, K. "What is Predictability for Real-Time Systems?". *Real-Time Sys.* 2, (4): 247-254. (1990).

[14] Audsley, N. C., Burns, A., Davis, R. I. and Wellings, A. J. "Integrating Fixed Priority and Best Effort Scheduling", *Proc. Workshop on Real-Time Programming*, Lake Konstanz, Germany, June 1994.

A Linear-Time Online Task Assignment Scheme for Multiprocessor Systems

Almut Burchard *

* School of Mathematics
Georgia Institute of Technology
Atlanta, GA 30332

Yingfeng Oh **, *Jörg Liebeherr* **, *Sang H. Son* **

** Computer Science Department
University of Virginia
Charlottesville, VA 22903

Abstract

A new online task assignment scheme is presented for multiprocessor systems where individual processors execute the rate-monotonic scheduling algorithm. The computational complexity of the task assignment scheme grows linearly with the number of tasks, and its performance is shown to be significantly better than previously existing schemes. The superiority of the assignment scheme is achieved by a new schedulability condition derived for the rate-monotonic scheduling discipline.

1 Introduction

Rate-monotonic (RM) scheduling is becoming a viable scheduling discipline for real-time systems. Through the years, researchers have successfully applied this discipline to tackle a number of practical problems, such as task synchronization, bus scheduling, joint scheduling of periodic and aperiodic tasks, and transient overload [4, 9]. This is done through developing various scheduling algorithms to cope with situations that are not covered by the rate-monotonic algorithm.

While rate-monotonic scheduling is optimal for uniprocessor systems with fixed-priority assignments, it is, unfortunately, not so for multiprocessor systems. In fact, the problem of optimally scheduling a set of periodic tasks on a multiprocessor system using either fixed-priority or dynamic priority assignments is known to be intractable [6]. Hence, any practical solution to the problem of scheduling real-time tasks on multiprocessor systems presents a trade-off between computational complexity and performance. Heuristic algorithms have been shown to deliver near-optimal solutions with limited computational overhead.

In this study, we are concerned with developing an efficient heuristic algorithm for scheduling a set of periodic tasks on a multiprocessor system. The general solution to such a problem involves two algorithms: one to schedule tasks assigned on each individual processor, and the other to assign tasks to the processors. In the following, we only consider multiprocessor systems where each processor executes the RM scheduling algorithm.

For the assignment of tasks to processors, one distinguishes *offline* and *online* algorithms. If the entire task set is known a priori, the scheduling method is referred to as being *offline*, otherwise it is said to be *online*. The task assignment scheme presented here belongs to the class of online algorithms.

Since real-time systems often operate in dynamic and complex environments, many scheduling decisions must be made online. For example, a change of mission may require the execution of a totally different task set. Or the failure of some processors may render a re-assignment of tasks necessary. In these scenarios, the entire task set to be scheduled may change dynamically, that is, tasks must be added or deleted from the task set.

Previous work on this problem illustrates the trade-off between computational complexity and performance of heuristic task assignment schemes. The complexity of an algorithm is given by the upper bound of the time required to schedule a set of K tasks. The performance of task assignment schemes is evaluated by providing worst case bounds for N/N_{opt}, where N is the number of processors required to schedule a task set with a given heuristic method, and N_{opt} is the number of processors needed by an optimal assignment. Bounds for the existing schemes are determined by $\lim_{N_{opt}\to\infty} N/N_{opt}$.

Davari and Dhall presented an online task assignment algorithm with a computational com-

0-8186-5710-3/94 $3.00

plexity of $O(K)$ and a performance bound of $\lim_{N_{opt}\to\infty} N/N_{opt} = 2.28$ [2]. Oh and Son developed two scheduling algorithm in [8]. The algorithms have a time complexity of $O(K \log K)$, and worst case performance bounds of $\lim_{N_{opt}\to\infty} N/N_{opt} = 2.33$ and 2.66, respectively. In both studies, the authors apply variants of well-known heuristic bin-packing algorithms where the set of processors is regarded as a set of bins [1]. The decision whether a processor is full is determined by a schedulability condition. All assignment schemes in [2, 8] are based on the sufficient schedulability condition for uniprocessor systems derived in [7] and its variants, e.g., [3]. Thus, these assignment schemes differ mainly in the choice of the bin-packing heuristic.

Our approach for developing a task assignment scheme for multiprocessor systems is different from previous work. Rather than increasing the level of sophistication of the bin-packing heuristic, we have developed a tighter schedulability condition that allows us to assign more tasks to each processor [1]. If the periods of tasks are sufficiently close, we could show that each processor can be almost fully utilized. Based on the new schedulability condition we present a novel online scheme for assigning task to a multiprocessor system. The complexity of our assignment scheme is given by $O(K)$ and the worst case performance bounds is shown to be $\lim_{N_{opt}\to\infty} N/N_{opt} = 1/(1-\alpha)$, where α is an upper bound for the load factor of any single task.

2 Task Model and Schedulability Condition

We assume that the real-time computer system consists of an homogeneous multiprocessor system and a set of K real-time tasks. The multiprocessor and the task set are characterized as follows.

A real-time task is denoted by $\tau_i = (C_i, T_i)$ $(i = 1, \ldots, K)$. T_i denotes the shortest time between two requests of task τ_i, and is referred to as the *period* of τ_i. C_i denotes the maximum execution time of task τ_i. Since we assume that the multiprocessor system is homogeneous the execution time of τ_i is identical on each processor. Each request for a real-time task must complete execution before the next request of the same task. Thus, in the worst case, the execution of τ_i must be completed after T_i time units. The period and the maximum execution time of task τ_i satisfy

$$T_i > 0, \qquad 0 \le C_i \le T_i, \qquad i = 1, \ldots, k$$

We will refer to $U_i = C_i/T_i$ as the *load factor* of the i-th task, and to $U = \sum_{i=1}^{K} U_i$ as the *total load* of the task set. We define α to be an upper bound for the load factor of any task, i.e., $\alpha \ge \max_{1\le i\le K} U_i$. ρ_n denotes the *utilization* of the n-th processor, that is, the sum of the load factors of the tasks assigned to processor n. Tasks are grouped into M classes, and only tasks from the same class can be assigned to the same processor.

Next we present a new sufficient schedulability condition for a processor that schedules tasks with the RM algorithm. The result, presented in Theorem 1, is a simple modification to the schedulability condition for uniprocessor systems by Liu and Layland [7]. Our condition yields a higher utilization of the processor if the task periods satisfy certain constraints. On a uniprocessor system, Theorem 1 does not provide a significant improvement for scheduling real-time tasks. For multiprocessor scheduling, however, we can divide a large task set into subsets in such a way that we can make use of the sharpened condition on all but possibly M processors.

The schedulability condition presented in the following theorem takes advantage of a special property of the RM scheduling algorithm. We show that we can increase the processor utilization if all periods in a task set have values that are close to each other. The proof of the theorem can be found in [1].

Theorem 1 *Given a real-time task set $\tau_1, \ldots, \tau_K$. For $i = 1, \ldots, K$, define*

$$S_i := \log_2 T_i - \lfloor \log_2 T_i \rfloor \qquad \textit{and} \qquad (1)$$

$$\beta := \max_{1\le i\le K} S_i \; - \; \min_{1\le i\le K} S_i \qquad (2)$$

A task set with $\beta < 1 - 1/K$ can be feasibly scheduled by the Rate-Monotonic algorithm if the total load satisfies (3). The condition is tight.

$$U \le (K-1)\left(2^{\beta/(K-1)} - 1\right) + 2^{1-\beta} - 1 \qquad (3)$$

Note that the condition given by (3) is tighter than the one given by Liu and Layland [7] under $\beta < 1 - 1/K$.

Corollary 1 *Given a set of real-time tasks $\tau_1, \ldots, \tau_K$. If the total load satisfies $U \le \max\{\ln 2, 1 - \beta \ln 2\}$, then the task set can be scheduled on one processor, where β is as defined above in (2).*

[1] The bin-packing problem is concerned with packing different-sized items into fixed-sized bins using the least number of bins [5].

3 An Online Task Assignment Scheme

Our new scheme is based on the schedulability condition of Theorem 1. The parameter used for the scheme, M, denotes the number of processors to which a new task can be assigned. Recall that tasks are divided into M classes. The class membership of a task τ is determined by the following expression:

$$m = \left\lfloor M\left(\log_2(T) - \lfloor \log_2(T) \rfloor\right) \right\rfloor + 1 \qquad (4)$$

Each processor is assigned tasks from only one class. Thus, at each processor the value of β as defined in (2) is bounded above by $1/M$. For each class, the scheme keeps one so-called *current processor*. If a new task from class m is added to the task set, the scheme first attempts to accommodate the task to the current processor for class m. A complete description of the algorithm for assigning a task $\tau = (C,T)$ is given in Algorithm 1.

In Algorithm 1, adding a new task $\tau = (C,T)$ is accomplished in the following manner. First, the class membership of the new task τ (Step 1) is determined. If τ can be added to the current processor of class m without violating the schedulability condition it is assigned to this processor. Otherwise, τ is assigned to an empty processor. If the load factor of τ is sufficiently small (Step 4), the processor to which τ is assigned becomes the current processor of class m (Step 5). If the load factor of τ is large, no other task will be assigned to this processor (Step 7).

Global functions:
curr(m) : Returns current processor for class m.
newproc() : Returns index of an empty processor.
Add $(\tau = (C,T))$
1. $m := \left\lfloor M\left(\log_2(T) - \lfloor \log_2(T) \rfloor\right) \right\rfloor + 1$;
2. **if** $(\rho_{curr(m)} + C/T \le 1 - \ln 2/M)$ **then**
3. $\quad \rho_{curr(m)} := \rho_{curr(m)} + C/T$;
4. **else if** $(C/T < \rho_{curr(m)})$ **then**
5. $\quad curr(m) := newproc()$; $\rho_{curr(m)} := C/T$;
6. **else**
7. $\quad x := newproc()$; $\rho_x := C/T$;
8. **endif**

Algorithm 1. Online Task Assignment.

The performance bounds of our scheme are given in Theorem 2 and Corollary 2. Corollary 2 states the asymptotic bound.

Theorem 2 *If a task set is scheduled by Algorithm 1, then the number of processors needed satisfies (5) and (6). Inequality (5) is tight if* $\alpha \le (1 - \ln 2/M)/2$, *and inequality (6) is tight if* $\alpha \ge (1 - \ln 2/M)/2$.

$$N < \frac{U}{1 - \ln 2/M - \alpha} + M \qquad (5)$$

$$N < \frac{2U}{1 - \ln 2/M} + M \qquad (6)$$

Proof. The schedulability condition used in Step 2 of Algorithm 1 enforces that at any instant, the load on all processors but the M current processors exceeds both $1 - \ln 2/M - \alpha$ and $(1 - \ln 2/M)/2$. □

Corollary 2 *Let* $\{\tau_i \mid i = 1, 2, \ldots\}$ *be a given infinite task set. Denote by* $U(k)$ *the sum of the load factors of the first* k *tasks. Denote by* $N_M(k)$ *the number of processors used by Algorithm 1, and by* $N_{opt}(k)$ *the number of processors used by an optimal scheme. If* $\lim_{k\to\infty} U(k) = \infty$ *then we obtain the asymptotic bounds (7) and (8). The bounds are tight.*

$$\lim_{k\to\infty} \frac{U(k)}{N_M(k)} \ge \max\left\{1 - \ln 2/M - \alpha\ ,\ \frac{1 - \ln 2/M}{2}\right\} \qquad (7)$$

$$\lim_{k\to\infty} \frac{N_M(k)}{N_{opt}(k)} \le \min\left\{\frac{1}{1 - \ln 2/M - \alpha}\ ,\ \frac{2}{1 - \ln 2/M}\right\} \qquad (8)$$

Proof. We obtain both (7) and (8) by passing to the limit in (5) and (6). □

From the derived bounds we see that the performance of Algorithm 1 is sensitive to the selection of M, the number of task classes. The asymptotic bounds in (7) and (8) improve for large values of M. However, M also determines the number of current processors, i.e., processors which are not fully utilized. Next we present a method for selecting an appropriate value of M for in equations (5) and (6).

Assume that the total load of the task set is known. To find the value of M that gives the best worst-case bound for the number of processors in (6), we fix the value of U in (6). Since the right hand side of (6) is a strictly convex function of M, we can calculate the unique minimum which is denoted by M^*:

$$M^* = \sqrt{2U \ln 2} + \ln 2 \qquad (9)$$

This suggests that we should choose $M \sim \sqrt{U}$. Then we obtain

$$U/N \ge \frac{1 - \ln 2/M}{2}(1 - M/N) \overset{U\to\infty}{\longrightarrow} 1/2 - O(1/\sqrt{U}) \qquad (10)$$

and hence

$$N/N_{opt} \leq 2 + O(1/\sqrt{U}) \qquad (11)$$

Similarly, we can minimize the right hand side of (5) over M and obtain that the optimal choice for M should be as close as possible to

$$M^* = \frac{\sqrt{U \ln 2} + \ln 2}{1 - \alpha} \qquad (12)$$

If we choose $M \sim \sqrt{U}$, we obtain with (5) the following bound for the average utilization at each processor.

$$U/N \geq (1 - \ln 2/M - \alpha)(1 - M/N) \stackrel{U \to \infty}{\longrightarrow} 1 - \alpha - O(1/\sqrt{U}). \qquad (13)$$

and N/N_{opt} is given by

$$N/N_{opt} \leq 1/(1 - \alpha) + O(1/\sqrt{U}) \qquad (14)$$

4 Average-Case Performance Evaluation of the New Scheme

While a worst-case analysis assures that the performance bound is satisfied for any task set, it does not provide insight into the average-case behavior of the assignment scheme. To gain insight into the average-case behavior of Algorithm 1, we conduct some simulation experiments.

Our simulations consider large task sets with $100 \leq K \leq 1000$ tasks. In each experiment, we vary the value of parameter M, the number of task classes. The task periods are assumed to be uniformly distributed with values $1 \leq T_i \leq 500$. The execution times of the tasks are also taken from a uniform distribution with range $0 \leq C_i \leq T_i/2$. Thus, α, the maximum load factor of any task, is given by $\alpha = 1/2$. The performance metric in all experiments is the number of processors required to assign a given task set.

We compare our scheme with the online assignment scheme by Davari and Dhall [2], NF-M. Recall that NF-M also has linear computational complexity. The outcome of the simulation experiments is shown in Figure 1. Since an optimal task assignment cannot be calculated for large task sets, we use the total load ($U = \sum_{i=1}^{K} U_i$) to obtain a lower bound for the number of processors required. The maximum number of task classes is set to M = 10, 20, 30, respectively. Each data point in the figure depicts the average value of 15 independently generated task sets with identical parameters. Note that for all values of M, our scheme gives superior performance over the existing one.

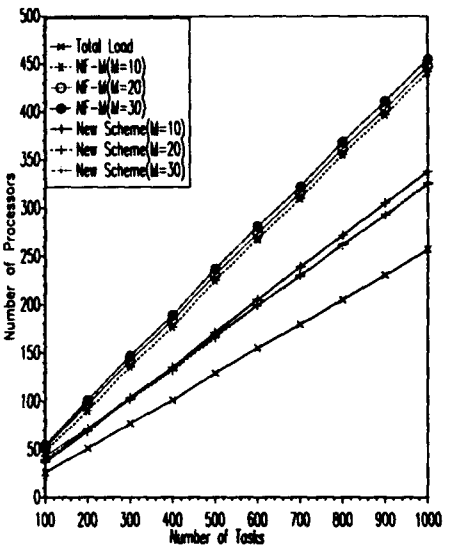


Figure 1: Task Sets with $\alpha = 0.5$.

References

[1] A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son. Assigning Real-Time Tasks to Homogeneous Multiprocessor Systems. Technical Report CS-94-01, University of Virginia, Computer Science Department, January 1994.

[2] S. Davari and S. K. Dhall. An On Line Algorithm for Real-Time Allocation. In *IEEE Real-Time Systems Symposium*, pages 194–200, 1986.

[3] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, January/February 1978.

[4] J. D. Gafford. Rate-Monotonic Scheduling. *IELE Micro*, pages 34–39, June 1991.

[5] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst Case Performance Bounds for Simple One-dimensional Packing Algorithms. *SIAM Journal of Computing*, 3:299–325, 1974.

[6] J. Y.-T. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, 2:237–250, 1982.

[7] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[8] Y. Oh and S. H. Son. On-line Task Allocation Algorithms for Hard Real-Time Multiprocessor Systems. Submitted for Publication.

[9] L. Sha and J.B. Goodenough. Real-time Scheduling Theory and Ada. *Computer*, pages 53–66, April 1990.

Session III: General

Chair: Mike Jones
Microsoft

Constructing a Heterogenoues Real-Time System

Sias Mostert
Department of Electrical and Electronic Engineering
Stellenbosch University
Stellenbosch, 7600, South Africa
email : mostert@firga.sun.ac.za

Abstract

The construction of a real-time system on heterogeneous hardware platforms, forces one to make choices on which programming language, operating system, development process and application programmers interface to use. The application (a micro-satellite) requirements state that the system must be dependable in a remote and harsh environment such as space. This paper will detail the choices made and the experience gained from living with the choices made in the development of a micro-satellite and its associated ground support. The emphasis is on simple solutions throughout. The simple solution is important for the verification and validation of the complete system.

1 Introduction

The construction of a real-time system on heterogeneous hardware platforms, forces one to make choices on which programming language, operating system, development process and application programmers interface to use. One has the option of going with main stream products or considering products with features of specific relevance to the application.

The application (a micro-satellite) requirements state that the system must be dependable in a remote and harsh environment such as space. This requirement in addition to the heterogeneous platforms found in the space and the ground segment of the micro satellite, complicates the choice of a development environment and target executable environment.

This paper will detail the choices made and the experience gained from living with the choices made in the development of a micro-satellite and its associated ground support. This project has been running for two years and is entering the software intensive phase.

Heterogenoues computing implies increased cost. It has however been shown in [1] that a careful application of hardware and software diversity can be cost effective.

The contribution of this paper is showing that the emphasis on simple solutions throughout provides an environment which is better suited for developing dependable real-time systems. Simple programming languages, operating systems, development lifecycles and application programmer interfaces all form part of the proposed solution.

The simple solution is important for the verification and validation of the complete system. Further more, the nature of the project environment causes us a 50% manpower turnaround every year [1]. The maintenance of the software must not take more than 20% of the man hours available.

The paper will begin by describing the hardware and software required for the application domain. Each of the areas in which a decision had to be made will be discussed in turn with all the options available, the final choice made and the reasons for doing so. The paper will close by reporting the experience we have had with our approach to constructing a heterogenoues real-time system.

2 The application requirements

The application software is for the space- and ground segments of a micro satellite constellation. The micro satellite is a 45cm cube box, weighing in at 50kg. The strict mass and power budgets places constraints on the flight control hardware which has a profound effect on the computing resources available. The real-time system of interest run on the space segment and the supporting ground segment.

[1] Graduate students are taken in every year and graduate after two years

0-8186-5710-3/94 $3.00

2.1 Hardware platforms

Traditional dependable hardware for use in space, is in the form of Triple Modular Redundancy (TMR). Each similar module is of high cost due to the reliability encased in it.

The less expensive alternative is to have heterogenoues hardware modules to increase fault tolerance and resistance to failure [1]. In our micro satellite the processors are an Intel 80C188EC, 80386SL and an INMOS T800.

In any space based system, the ground support adds another dimension to heterogeneity, because it being accessible for repair and the power budget infinite, it does not have to be the same architecture as the space segment.

2.2 Software composition

The levels of reliability required from software in the application domain ranges from ultra reliability to acceptance of an error once per 30 days. The table in fig 1 summarizes the software functions and the reliability required.

3 Languages

There has been numerous reviews and comparisons of languages suitable for real-time applications [2, 3]. It all cases it was argued that mainstream languages such as C, Pascal, Ada, Modula-2 etc. were not suitable for the construction of real-time systems.

From the available commercial real-time operating systems [4] one can deduce that C and C++ are being used extensively for real-time applications. From our experience C is not more than a glorified assembler and is to be avoided for dependable software. In the favor of C is that is by far the most widely implemented compiler on different hardware platforms.

The following requirements dictated our language choice:

1. Most of the Software Engineers are Electronic Engineers with limited training in Software Engineering.
2. The project staff join the project for at most two years at a time.
3. Most of the flight software has to be very reliable.
4. Most of the flight software is to be maintained across a five year time span.

The language of choice for the space based reliable software is Modula-2 with small sections of Assembly language in such cases as working without a stack before the memory integrity has been determined.

Modula-2 provides the following advantages.

1. It is more readable for people with less software training. This aides in maintenance and the fact that most of our staff have limited training.
2. It is stricter than C and C++ in its syntax which makes it more difficult to induce unintended errors. The stricter syntax also aides in the verification and validation process, making it easier to create reliable software.
3. Compilers are available for all our microprocessors.

4 Operating System

The requirements placed on the Operating System is different for the space and ground segments. The space segment hardware resources are costly and bounded by power and mass budgets. The ground segment hardware resources are for all purposes unbounded.

The requirements for the space based operating system are support for process dispatching, inter process communication, synchronization, loading and unloading of process sets, support for interrupt handling and time functions. The implementation of these requirements on different processors leads to the choice between software diversity and -homogeneity.

The cost effective use of software diversity is explained in detail in [1]. It amounts to using software diversity on the kernel level where it can be afforded and no software diversity on the application level, where it is expensive. Due to the fact that our software on the kernel level is reloadable in the final space segment component, it was decided in the interest of development time to choose one operating system kernel for the current software support.

The space segment processors require an efficient Operating System Kernel in order to make best use of the resources. There are many such kernels on the market [4], which all support priority based scheduling.

Deadline driven scheduling is optimal when compared with rate monotonic [5], and deadline driven scheduling specifies end to end deadlines more succinctly.

Software component	Hardware support	Reliability required
Boot loader	Fusible link PROM	No errors
Default application	EPROM	No errors
Household and Integrity Tasks	SRAM with EDAC	One error per 30 days
Operating System Kernel	SRAM with EDAC	One error per 30 days
Device drivers	SRAM with EDAC	One error per 30 days
Fine ADCS Application	SRAM	One error per 30 minutes
Bulletin board Application	SRAM with EDAC	One error per 30 days
Experimental Software	SRAM with EDAC	One error per orbit

Figure 1: The software functions and the reliability required

It was decided to go with a simple kernel supporting deadline driven scheduling [6]. The specific paradigm was proposed by [7] and we are using an implementation done by [8], calling it RTXN.

The requirement for the uploading and unloading of application task sets, are to be supported through the concept of Virtual Machines (VM). Each VM will represent a hardware resource or percentage of hardware resource. The executing code of a VM can be completely replaced without affecting any other VM. Multiple VMs are to run on each processor in the space segment with each VM representing a percentage of the underlying resource available. Within each VM the RTXN kernel will be executed on which the application task set can be executed with real-time constraints guaranteed.

The ground segment processors are in such abundance that it was decided to support the resource adequate paradigm [9]. Each application on the ground would run on its own processor (80x86 PC). The penalty to pay for this approach is a communication mechanism which must be maintained between processors.

5 Application Programmers Interface

In order to provide access to a unified architecture across space and ground segments an Application Programmers Interface (API) is required. This API is called the SUNSAT-API (SSAPI) and is defined as the simplest subset of functions required for the parallel execution of processes in a Hard Real-Time environment.

The SSAPI supports only those functions as mentioned in section 4 and is described in detail in [10]. The same SSAPI is also supported on the ground segment, which enables the Application Software Engineer to concentrate on the task and not the idiosyncrasies of the different kernels on the different processors.

6 Software development processes

The diverse range of reliability requirements on the software necessitates a flexible approach to software development. We have opted to support the waterfall development model with the extension of rapid prototyping before the software requirements phase is complete.

The flexibility is built into the waterfall model by requiring the outputs of the different stages to be checked only for specific types of software. See table 2 for the types of software and the associated checks which must be performed on the output of each stage.

The are strict standards in place for the following outputs:

1. Software requirements document.
2. Software design document.
3. Software testing document.
4. Software coding standards.

In addition the design languages are pinned down to enable faster inter- engineer communication. The design languages can be split in two groups ie. structural and behavioral. The table in fig 3 lists the options available under each of the groups.

The structural design languages are aimed at arriving at reusable software and specifies the way in which the software is to packaged. The behavioral design languages specify how the software is going to behave in multiple dimensions, which include the state, the execution flow and the timing characteristics.

	Development phase				
Type of software output	Requirements	Concept design	Detail design	Coding	Install
Hardware debugging	*			*	*
Subsystem testbed	*			*	*
Hardware demonstration software				*	*
Flight software	*	*	*	*	*
Ground station software	*	*	*	*	*
Porting an existing software	*	*	*	*	*

Figure 2: The application of the Waterfall development lifecycle to various types of software

Structural design languages	Behavioral design languages
Modular decomposition	Flowcharts or pseudo code
HOOD (High Order Object Oriented Design)	Statecharts
Software topology	SSAPI dataflow diagrams
Hardware topology	

Figure 3: Structural and Behavioral Design languages

7 Results

The results up to date is promising. We have progressed one year with the initial testing of the methodology on small pilot projects within the major project and found the following:

1. The choice of Modula-2 (and Pascal substituted in some cases) as programming language has in fact enabled the new intake of engineers to get up speed in a shorter period of time.

2. The boot loader code has gone through the requirements, conceptual and detail design phases, with formal checks on the output of each phase and we have a greater confidence in its correctness than any other piece of software on the project.

3. We have found that the proper execution of the requirements and conceptual design phases did take about 70% of the time, but that the coding was in fact a formality.

4. For non-ultra reliable software we found that it is adequate to proceed with the design until the module interface level. The additional time spent on detail design did not provide any additional benefits in saving time for the reliability required.

5. The SSAPI is proving to be of great benefit as more than one person is working on the software for the same processor. The common design language has definitely saved on man hours supporting the execution environment.

6. The resource adequate strategy for the ground station is proving that every single application developer can go ahead regardless of any other person. This facilitates true concurrent engineering and simplifies the checking for adhering to hard real-time requirements.

7. The choice of going with a simple, non-commercial kernel has not proved us wrong yet. The simplicity ensures that one person can understand the complete kernel which ensures complete transparency through the kernel when implementing on a specific hardware platform.

8 Conclusion

We have had to make many choices for implementing a Real-Time System on a heterogeneous platform. Neither the programming language nor the operating system kernel is mainstream, but we believe that the penalty paid for it (lack of support)will provide the return in ease of maintenance in the long run.

We have finished our initial investigation into the suitability of the paradigm explained and we are producing software for our first integration of the Engineering model based on the paradigm selected. On completion, the paradigm will be evaluated before the actual flight software is produced in the latter part of 1994 and the beginning of 1995.

Acknowledgements

The SUNSAT team responsible for the pilot projects include Pieter Bakkes, Mike Blankenberg, Jurie du Toit, Herman Gouws, Rein Brune, Ian de Swardt and Louis Jordaan.

References

[1] Sias Mostert. Hardware and Software Diversity for a Fault Tolerant Space Data Management System. Technical Report SUDEE-2/93, University of Stellenbosch, Stellenbosch, South Africa, December 1993.

[2] Wolfgang A. Halang and Alexander D. Stoyenko. *Constructing Predictable Real Time Systems.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991.

[3] Phillip A. Laplante. *Real-time Systems Design and Analysis.* IEEE Press, 1993.

[4] Martin Timmerman. RTOS Issue II. *Real Time Magazine*, 93(4):6–104, December 1993.

[5] C. Liu and J Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20:46–61, 1973.

[6] Richard K.J. Henn. Feasible Processor Allocation in a Hard-Real-Time Environment. *Real-Time Systems*, 1(1):77–93, June 1989.

[7] Kevin Jeffay. *The Real-Time Producer/Consumer Paradigm: Towards Verifiable Real-Time Computations.* PhD thesis, University of Washington, September 1989.

[8] Marius Ackerman. *A kernel for temporally correct reactive systems.* M thesis, University of Stellenbosch, 1993.

[9] Harold W. Lawson. Cy-Clone: An Approach to the Engineering of Resource Adequate Cyclic Real-Time Systems. *Real-Time Systems*, 4(1):55–84, March 1992.

[10] Sias Mostert and Pieter Bakkes. An architecture supporting dependable real-time systems. Technical Report SUDEE-1/94, University of Stellenbosch, Stellenbosch, South Africa, January 1994.

Using SDL in Embedded Systems Design: A Tool for Generating Real-Time OS pSOS based Embedded Systems Applications Software

Ye Huang Michael Hughes

R&D and Consulting
Anonymix Inc. Los Gatos, CA95030
E-mail: yhuang@anonymix.com

Abstract

In this paper, we present an efficient method using Specification and Description Language (SDL) for designing and implementing real-time embedded systems. We also discuss the implementation of a companion kernel for a SDL based Design Tool (SDT) [1] CASE tool environment to generate real-time OS based pSOS [2] multi-tasking application software by applying defined mapping translation rules. Since SDL is a formal specification and description language, with the CASE environment SDT support, the major part of a system can be analyzed, simulated, verified, and validated at early stages during system development. The concurrency due to the multiple concurrent state machines in a system is preserved in target run-time environment. Because the SDL described system uses message-passing, a distributed version can be relatively easy to derive from. To emphasize the proposed method using SDL without dramatically compromising the memory and response time (speed), we show the results obtained from pSOS implementation of an AccessControl system. We also outline some areas that we are continuing to work on.

1 Introduction

1.1 Eveloving Software Development Methods for Embedded Systems

Software system design and development has long been regarded as art rather science or engineering. The term *Software Engineering* was formally coined at a NATO conference in 1968, which signalled that systematic and engineering displinary methods and techniques should be employed to produce quality and cost-effective software system.

Significant improvements have been made since then in many fields that ensure the software systems with better quality in a controlled and cost-effective manner thanks to the research efforts in the software engineering, methodology, and computer-aided software engineering envrionment. Many fairly sophisticated and large CASE tools exist, such as Teamwork, Statemate, and ER-Designer [4], which assist the system designers and developers in continuing improving the quality and productivity of documentation and code.

The traditional real-time embedded systems development methods emphasize memory and speed by handcoding the software systems in assembly languages.

With the increasing performance of processors and hardware devices, memory and speed are less important. Thus the hardware technology has led to widely use mixed development of software systems. The majority of the software systems is developed on the host development environment in high-level programming language, such as C or C++. Assembly code is still needed for some certain time-critical tasks and device drivers and interrupt routines.

Object-Oriented techniques and methodologies further improve the real-time embedded systems design and implementation in terms of software reusability, quality, documentation, and classification [5]. Object-Oriented CASE tool environements have played major role in this new trend.

[1] SDT is a trademark of TeleLOGIC AB.
[2] pSOS is a trademark of Integrated System Inc.

0-8186-5710-3/94 $3.00 © 1994 IEEE

1.2 Current Methods with CASE Tool Support and Our Approach

The following steps are not uncommon for the current embedded systems design and implementation:

Step 1 : Requirement analysis

Step 2 : Specification and design

Step 3 : Implementation

- Host simulation and debugging
- Target simulation and debugging

Step 4 : Testing (and may iteratively goto *Step 1*)

There are a number of problems among the existing CASE tools used today, such as informality, non-analysibility, and non-validatability, to name a few.

Aside from the confusion of different notations for those methods, most of them are based on preprotary formalisms and lack mathematic foundation needed for system consistency analysis, verification, and validation.

There tends also to be a gap between design and implementation. Design is automated to certain degree, but implmentation is conducted separately and often manually. The implication is that the implementation is not ensured due to the discontinuity between the design and implementation.

The next factor that hinders the software systems prototyping and implementation for the embedded systems is coding and debugging. Even using increasingly sphisticated host development environments, developers are often overwhelmed in coding at the API level (C or C++ lanaguage). The debugging only occurs after downloading the cross-compiled system onto the target. Then the usual nightmare is that significant time is spent on deep-level dubugging. Though, sometimes, host simulation and debugging are provided,

Our primary focus is try to emphasize the suitability of SDL in embedded systems specification and design. Secondly, we tried to bridge the gap between design and implmentation by mapping out rules and algorithms to automatically generate the final target environment code.

SDL is a well-known specification and description language standardized by ITU [3] as Z.100 [1] [2]. Its success and popularity are still growing, especially in the telecom and datacom sectors, especially in Europe. Moreover, this proven technology for the systems specification and design has been evolving gradually. SDL92 is more object-oriented.

The methodology for using SDL and SDT in designing and implmenting software systems for the real-time embedded systems is similar to that for the telecom software system as outlined below

Step 1 : Requirement analysis (Message Sequence Chart)

Step 2 : System Specification (SDL)

Step 3 : Semantic and dynamic system analysis

Step 4 : Simulation, verification, and validation

Step 5 : Code generation

Step 6 : Testing (conformance testing)

Step 1 through *Step 6* are supported by SDT. Various SDL techniques have been studied and exeprienced successfully, and broad literature can be found [8] [7] [1] [?]. We will mainly discuss the code generation for the embedded system since the software systems for the embedded systems have some unique characteristics that are often not required by other types of applications, namely, time-criticality, response time, and fast I/O processing.

To meet such kinds of operational requirements, two types of philosophy are instrumental in the design process:

- Build software system on top of a commercial real-time operating system
- Build software system along with a self developed kernel

Depending upon the applications and desired embedded system configuration, these two approaches are alternately used or interleaved. For instance, a distributed embedded system's front end agent does not have real-time OS, but the some large nodes and host nodes employ the real-time OSs.

When building around an existing real-time OS, we describe a model and SDT to automatically generate the target code that fully utilizes the real-time OS scheduling, memory management, and time management capabilitie. For the latter approach, we augument the model by adding additional scheduling algorithms to the generic SDT kernel to fit a particular application.

Critics have been questioning automatic-code generation in terms of the quality of code, size of the code,

[3] ITU stands for International Telecommunications Union previously called CCITT.

speed of systems with regard to response time to critical device requests and interrupts. We have conducted some initial analyses based on an AccessControl system generated for a M68K systems rnning pSOS.

The following are major highlights from our experience:

- Code size overhead: 1.3-1.5:1
- Code quality: No debugging efforts for the majority of system except for the interrupts and device driver part and interface part during the final intergation.

Systems maintenance and reusability are much easier now due to the fact that we are dealing with the most part of the software system at a high level - specification and design.

Research efforts are onging on the use of SDL for system specification and design. Some researchers have proposed notations and supporting systems trying to bridge the gap between requirement analysis and specification. To a certain degree, the design specification can be automatically produced from requirement specification [9].

Obviously, this method along with the tool support provides us with better system maintainability, tracibility, and other benefits, such as testibility.

2 SDL based Approach Overview

2.1 Overview

The SDL based software modelling technique is not new. The language was standardized in 1988 as Z.100 and it is still evolving. The current SDL92 is a superset of the SDL with object-orientation extension, which is also called object-oriented SDL.

SDL was created for modelling large and complex distributed real-time telecommnication software systems. Substantial experiences have been gained in using SDL for the system specification and validation. In many cases, real implementation code was generated and deployed.

Many methodologies based SDL have been proposed and employed [1] [7] [6] for the systems specifications and design. We will omit the mature and common set of the procedures and stress those issues specially related to embedded systems, such as target code generation and systems integration.

The following steps are suggested for modelling and implementing an embedded system:

Step 1: Define the system architechare (configuration of hardware and software);

Step 2: Define the interface between the software system and hardware system in terms of interrupts and device drivers;

Step 3: Model software system using SDL and hardware interface in the environment either using SDL or unix processes as separate systems;
(Note: the hardware device drivers and interrupt routines in assembly level language can be developed simutanously.)

Step 4: Use the SDL modelling techniques to prototype, simulate, validate, and test (conformance test) the system under design;

Step 5: Generate target code when *Step 1* through *Step 4* completed by compiling and linking the proper shared library, i.e. library for taregt real-time OS or without target real-time OS;
(Note: test suites also can be generated automatically for unit or system test.)

Step 6: Use proper target debugging environement to test the integrated embedded system on the target systems; and

Step 7: Depending on necessity, perform some manual code optimization for certain critical tasks.

In this suggested method, *Step 1* through *Step 4* are relatively well understood and deployed. *Step 5* through *Step 7* are of interest to us here in paper.

2.2 Tool configuration for target code generation

In SDL, system behavior is described by state machines each representing a SDL process. Communications mechanisms among the blocks and processes are realized by signalling (i.e. sending and receiving messages). The time constrains are imposed by setting timers. The code generated from SDL design implements a complex finite state machine (FSM) comprising many concurrent FSMs communicating via signals.

Figure 1 shows the SDT tool configuration with extension for generating pSOS target code.

For many embedded systems, there are some special issues, such as concurrency (multi-tasking), synchronization, fast multiple I/O processing, and interrupt handling.

SDT has a generic kernel for both host environment simulation, validation, and verification and target envrionment with some minor patching in clock

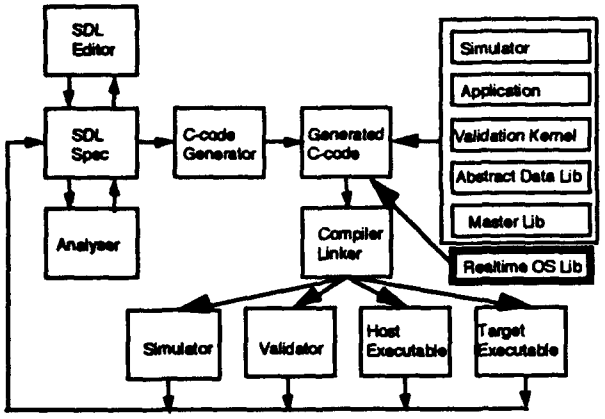


Figure 1: SDT tool configration with extension for generating pSOS real-time OS based software systems: dashed-lined component

and memory management handling. The SDT scheduler is priority oriented with granularity to the state transition. The synchronization among processes is realized through signalling.

For embedded systems without real-time OS, a generic SDT is privided by SDT CASE tool. The generated code from the SDL design can run with the SDT kernel. If there are special domain requirements to the OS, the current SDT scheduling algorithm can be modified, such as time-slicing and pre-emption.

When generating code for embedded systems with Real-Time OS support, such as pSOS, the system designed in SDL will be transformed into the designated target real-time OS based software system. The mapping rules and transformation algorithms are maintained in the pSOSlib. The current implementation maps the processes in SDL to a real-time OS tasks and signals to messages.

3 The Mapping Model for Generating Real-Time OS based Code

The design for embedded systems without real-time OS support has been briefly discussed in the previous section. We focus on the mapping rules and translation algorithms to generate code for pSOS target environment here [4].

[4]Note:in the discussion followed, although pSOS is used, the mapping rules are similar when applying them to other real-time OS environments such as VxWorks.

3.1 SDL Systems Semantics and Its Execution Model

For continuity, we only revisit the definition and key aspects of the model and its semantics. Refer [1] and [8] for detailed description. SDL is based on the concepts of a system of communicating extended finite-state machine (CEFSM) that comunicate with one another and their common environment by signals in an asynchronous manner via possibly delaying communication paths. These signals are buffered on arrival at a process.

An SDL specification represented by a execution model comprises seven meta-process types. These processes are Communicating Sequential Processes (CSP), and thus they communicate using synchronous events. These meta-processes are: (1) **system** The only instance of this meta-process type creates all other meta-processes and maintains them; (2) **delaying-path** One instance of this process type for each communication path between processes of the SDL system; (3) **global-time** The entity knows the current global time; (4) **view** The only instance of this process type keeps track all viewed variables; (5) **sdl-process** An instance of this meta-process type represents a process; (6) **input queue** An instance of this meta-process type represents the input queue of a process; (7) **timer** An instance of this meta-process type represents a timer of a process.

3.2 Mapping Rules and Trasformation Algorithms

When formulating mapping rules and transformation algorithms to transform a SDL system onto a real-time OS system, such as pSOS, the rules to follow are: (1) preserve SDL dynamic semantics; (2) identify RTOS objects to realize SDL objects; (3) implemetn additional software to augument the portions that RTOS does not support or best match.

In our case, we map a SDL process to a pSOS task, a input queue to a pSOS message queue. Due to some deficency between pSOS time management services and SDL's, we implemented our own timer management. When, some improvements on the time services made, we are to use pSOS's for a tigher integration which will reduce code size and fully utilize the pSOS kernel features. Because of the richer SDL communication properties, such as sending signal without specifying receiver PID, we implemented a relative concise system.

The view meta-process was not implemented considering its side-effect on the code modularity and reusability.

3.3 Opimization Issues

To optimize the performance, we deliberately require the SDL user to specify the receiving process PID. The idea is that no routing function needs to be performed to search the receiver in a embedded environment. However, we did implemented the full-SDL routing semantics for those who desire to use.

4 Experiment Results and Analysis

We have used a classic example in the SDL literature [8] [6] AccessControl system to perform some prelimary performance analysis. Following are the major points to be discussed:

- Outline of system requirements
- System description in SDL
- Measurements of generated pSOS application code
- Runtime performance data analysis, such as system call frequency, input queue length, and response time

At the end of this section, some comparison data analysis is expected And performance analysis for generated system with or without routine module is given. We also report how the performance was improved by introducing a static-dynamic combined mechanism to optimize the performance and maintain the SDL semantics.

5 Conclusion and Future Work

We think the method using SDL with the companion tool to a large extent will improve the quality, lead time, and cost for engineering embedded systems.

However, there are other issues and areas to be further studied in this direction. The following list is intended to shed some light on what we feel that should be focused:

- Automatic generating distributed applications for multi-processor architechure;
- Process-task grouping rules and realization
- Code optimization

Acknowledgements

Special thanks go to our colleagues Olle Hydbom and Jan Karlsson at TeleLOGIC for their kind support and work in making changes in code generator. We would like to thank Dong Parker and Manish Vadeya from Integrated System for the discussions with them and their comments and suggestions

References

[1] Internaltional Telecommunications Union, "Functional Specification and Description Language (SDL) Criteria for Using Formal Description Techniques (FDTs)", Blue Book, IXth Plenary Assembly, 1988

[2] Internaltional Telecommunications Union, "CCITT: Z120 Standard", Blue Book, IXth Plenary Assembly, 1988

[3] Christina Groove, " Par-SDL code generation for transputers," *SDL Forum 1993*, Gernmany, Oct, 1993, pp. 110–120

[4] Alfonso Fuggeta, P. Milano, and Cefrifl, " A Classification of CASE Technology", *IEEE Computer*, *Vol.28, December*, 1993, pp. 25–45

[5] Grady Booch, "Object-Oriented Design with Applications", The Benjamin & Cummings Publishing Co., 1992

[6] Rolv Brak and Oystein Haugen, "SDL Methodology", 4th SDL Forum Proceedings, Oct. 1993, Germany, pp. 111–222

[7] Roberto Saracco, J. R. W. Smith, and Rick Read, "Telecommunications Systems Using SDL", North Holland, 1993

[8] Frenc Belina, D. Hogref, and A. Sarma, "SDL with Applications from Protocol Specification", Printice Hall, 1992

[9] Haruhisa Ichikawa, Masaki Itoh, June Kato, and Masashi Shibasaki, "SDE: Incrementatl Specification and Development of Communications Software", *IEEE Trans. on Computers*, Vol. 40, No. 4, April 1991, pp. 553-569

Practical Formal Development of Real-Time Systems

S. P. Bradley W. D. Henderson D. Kendall A. P. Robson

Department of Computing
University of Northumbria at Newcastle
Newcastle-upon-Tyne, NE1 8ST, UK

Abstract

The complexities of real-time systems are such that it is often thought necessary to give a formal justification of their correctness, especially if they are to be used in a safety-critical environment. In this paper we describe our work on a formally based design method for real-time systems which allows the timing aspects of a concurrent system to be mathematically described and verified, as well as semi-automatically implemented. Our design language, AORTA, is a timed process algebra, with features to ensure that all designs can be implemented. A predictable real-time kernel is also described, which is used in the construction of a system from an AORTA design, and which allows the timing of the implementation to be verified.

1 Background and motivation

There is much existing work on methods for real-time systems, both on the theoretical aspects of verifying the correctness of a real-time system [1], and on the practical ways of guaranteeing performance via a real-time kernel [2], but little that links the two. If practical formal techniques are to be found for real-time systems, then both high-level, more theoretical aspects must be considered as well implementation performance issues such as scheduling. There is some work which attempts to link the higher and the lower level, such as the implementation of formal models by compilation of (untimed) LOTOS [3, 4], or the overall system design methodology of the (non-formally based) MARS project [5], but we are not aware of any work that addresses the practical design, implementation, and formal verification of a time-critical system. In response to this apparent lack, we have developed a formal design language based on process algebra called AORTA (Application Oriented Real-Time Algebra) [6], with the specific aim of providing a complete route from a (timed) formal specification to a verified implementation.

The first goal of our project has been to provide a means of producing a real-time system from its formal expression in AORTA, as most of the existing theoretical work covers the verification of a design with respect to a formal specification. Work on real-time kernels has made advances in ensuring that processes will get through their work as quickly as possible. Sometimes, however, this is at the expense of making the scheduling arrangements too complex to be able easily to provide reliable predictions about the performance of an interacting set of processes. Whilst priority-based scheduling algorithms may be provably optimal, it is not always optimality that is important — in particular, predictability of time-critical systems can be crucial. On this basis we have reverted to a very simple yet predictable fixed time-slice round-robin scheduler, so that timing is easier to predict, as the performance of each process does not depend on the performance of others except at explicit communication or synchronisation points. The efficiency sacrificed in using such a scheduling mechanism is balanced with the reduced cost of developing a verified system; as hardware costs are relatively low compared with development costs, we feel that this tradeoff is often justified.

The kernel also provides sound, safe and predictable communication primitives, based on Ada style synchronous communication, which correspond directly to the communication constructs in the AORTA design language. Together with a timeout facility, this provides a direct route to implementation, by C code generation from the AORTA design. Although the AORTA design only deals with the timing and intercommunication of the processes, the sequential code within a process is included in a manageable way, and the timing of non-generated code is verified by a combination of bounds on processing time of the code and the processing distribution figures available for the kernel in [7].

0-8186-5710-3/94 $3.00

2 The AORTA design language

Timed process algebras are widely known, but are usually used for modelling or specifying real-time systems, rather than designing them. Our language, whilst having some features in common with timed (and untimed) process algebras, is distinguished by its implementability. There are two main points of difference, the first being that the number of processes within a system is constant throughout the lifetime of the system, making processor allocation, and hence computation times, easier to verify. Secondly, all timings in the design can be expressed as upper and lower bounds, rather than exact figures, as in reality bounds can usually be given, where precise figures may not exist. It is this representation of time bounds which is most problematic in existing timed process algebras.

In order to keep the model of the timing tractable, data within a system is not represented in AORTA, with all computation being represented only by the (bounds on the) amount of time required to complete it. Within each process communication is represented by the name of the gate on which communication is to take place, and bounds are placed on the amount of time between both sides of the communication being ready, to the communication actually taking place. Processes are written in a simple equational format, similar to that used in many other process algebras. A process written

```
Convert = in.[100,150]out.Convert
```

will wait for communication on its `in` gate, before doing some computation which lasts for between 100 and 150 milliseconds (any time units, discrete or dense may be chosen for a design — [0.1,0.15] would be an equally valid expression) and offering communication on its `out` gate. Once this second communication has taken place the process will start again waiting for an `in` communication. This is how a process which accepted temperature data and converted it to a different format would be expressed. The bounds on communication times are given by a separate function which takes a gate identifier and returns a time interval.

A choice construct is provided, similar to the + of CCS and the `select` statement of Ada, which allows several possible communications to be offered. The future behaviour of the process depends on which is completed first. Timeouts may also be defined, so that if none of the communication choices offered are taken up within a certain time, then control passes to another branch. Again, exact figures are not usually available for occurrences of timeouts, so bounds are used instead. If our `Convert` process is to accept input or allow its conversion mode to be changed, this would be written

```
Convert = in.[100,150]out.Convert
          +
          mode.[300,400]Convert
```

where the reconfiguration procedure takes between 300 and 400 milliseconds. If the data offered on the `out` gates is also to be kept up to date then it may need to be refreshed every 1.5 seconds or so, which is achieved by adding a timeout to the `out` communication:

```
Convert = in.[100,150]
           (out.Convert)[1450,1550>Convert
          +
          mode.[300,400]Convert
```

where 1450 and 1550 are estimates on the bounds which can be placed on a timeout of about 1500 milliseconds.

The last construct which can be used in the definition of individual processes is a data-dependent choice, used where the flow of control of the process depends on the value of some data in the system. Data is not modelled in AORTA, so this is essentially a nondeterministic choice as far as the process algebra is concerned. The choice between two possible behaviours is represented by ++, so that if our `Convert` process is to give a warning if the value that it finds is outside a certain range, this would be written

```
Convert = in.(Convert2 ++ warning.Convert2)
          +
          mode.[300,400]Convert
Convert2 = [100,150]
            (out.Convert)[1450,1550>Convert
```

A system usually consists of the parallel composition of two or more processes; this is represented using the traditional process algebra bar |, with a connection set showing pairs of gates which may communicate. This explicit connection of gates allows for a more efficient implementation, and simplifies verification. The connection set is represented by pairs of gate identifiers written in angle brackets after the processes of the system. A plant control system incorporating the `Convert` process with a `Control` process and a `Datalogger` process, is written as follows:

```
Tempsys =
( Control | Convert | Datalogger )
<(Control.changem,Convert.mode),
```

```
(Control.temphigh,Convert.warning),
(Convert.out,Datalogger.getdata),
 connections between Control and Datalogger
>
```

The ordering of the pairs of gates is not important, and not all gates of the processes need be connected: those left free will have to communicate with the environment, like the **in** gate of our **Convert** process. Figure 1 gives a diagrammatic representation of **Tempsys**.

Most features of typical small embedded systems can be designed using this language: resource contention can be handled with the choice construct, and polling loops with a timeout. As well as allowing implementation, AORTA has a formal semantics given in terms of a timed transition system, which allows formal reasoning to be done about the design, and the possible application of model-checking techniques such as [8, 9]. Space does not allow us to go into the details of the semantics here, but see [6] for more details — it remains now to show how AORTA designs can be implemented in practice.

3 Implementation and the kernel

As the main point of the AORTA design language is its implementability, we outline here the kernel which we have written which allows AORTA designs to be verifiably implemented. We mentioned earlier that we have adopted a very simple approach to scheduling in order to be able easily to verify the performance of each of the processes. This is achieved by using a fixed time-slice round robin scheduler, where a fixed schedule of processes is executed on the kernel at a fixed frequency, so that each process has a guaranteed amount of processing time per unit real time, unaffected by the performance of the other processes. For a given amount of processing time required, bounds can be put on the amount of real time required, so that the timing of a piece of sequential computation can easily be verified given the processing requirements of that computation. Bounds on the computation time required for a piece of code can be found using techniques such as described in [10].

At each scheduling point, the kernel checks through the list of connected gates to see if there is a pair which is ready; if there is then it effects the communication, signalling to the processes involved that it has taken place, and disables communication on gates which were in choice with the successful gates. It then looks for possible external communications (on gates that are left unconnected) before looking through the list of timeouts to see if any have exceeded their time limit. By checking for communications and timeouts at every reschedule, bounds can be placed on the time for a communication to take place once enabled, and for a timeout to come into effect.

Communication primitives are offered by the kernel as C functions which are called from the processes. The calls to the kernel are generated automatically from the design, along with the parameters of the kernel, such as the number of processes, and the gates which are to be connected. Details such as the code to be executed as part of a computation delay, the data to be passed in the communication, and the conditions for a data-dependent choice are attached as annotations to the design. They have no interpretation in the formal semantics, where they are viewed as comments, but they allow the code to be included in the correct place without having to edit the code generated from the design. Putting the kernel together with the generated code allows an implementation to be generated automatically from an annotated design. The pieces of sequential code still have to be hand constructed, but once written, their timing, and hence the timing of the whole system can be verified.

4 Current and future work

The work on providing a route from design to implementation is complete, so that a system can be built automatically from its design. Although all of the verification methods are manually available, they have not yet been integrated into a single tool. The verification of an AORTA design will be addressed soon, but there is currently a simulator tool, which allows a design (including its timing) to be tested out by a user as the first step in a verification process. It is hoped that existing formal verification methods (such as [8, 9]) may be applicable. Figure 2 shows how the work fits together: arrows going downward represent implementation, arrows upwards verification; solid arrows indicate currently available routes, automated where appropriate, and the dashed arrows represent possible future pieces of work. Other implementation routes may be the subject of future work, such as distributed implementations or kernels based on other scheduling mechanisms. One particularly interesting piece of future work would be the integration of existing formal methods for developing sequential code (such as Z [11] or VDM [12]) with AORTA, so that the timing of a system and its functional correctness could be verified in a unified way.

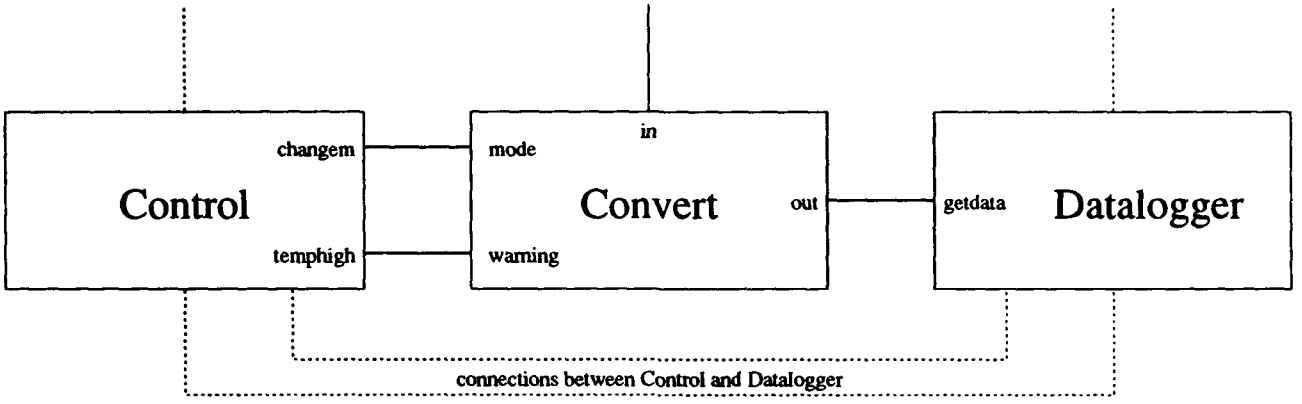


Figure 1: Connectivity of **Tempsys**

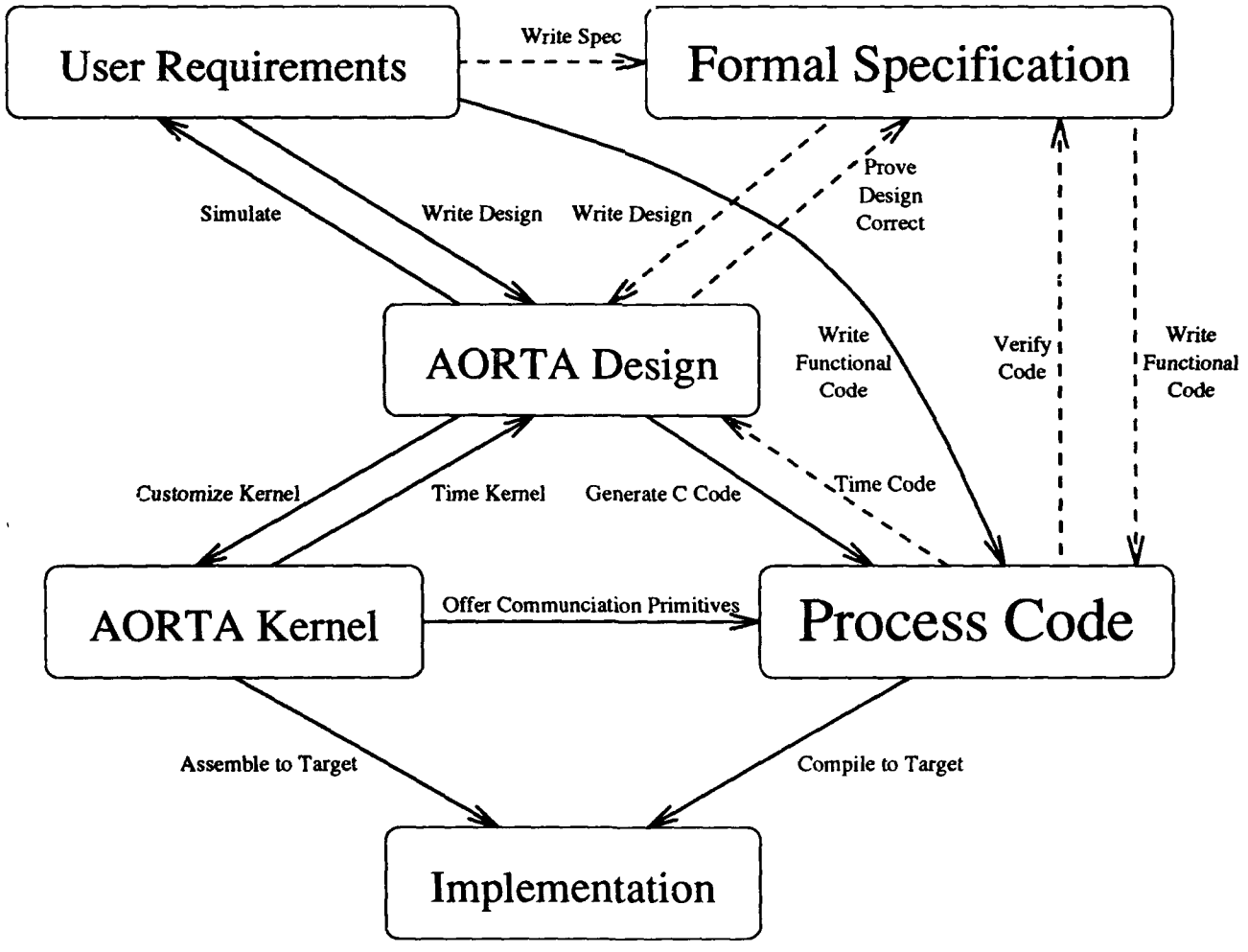


Figure 2: AORTA within a real-time design methodology

Acknowledgements

The authors would like to thank the University of Northumbria at Newcastle and Northern IT Research for their financial support, and the anonymous reviewers for their comments.

References

[1] J S Ostroff. Formal methods for the specification and design of real-time safety critical systems. *Journal of Systems and Software*, 18(1):33–60, April 1992.

[2] A Burns. Scheduling hard real-time systems : a review. *Software Engineering Journal*, 6(3):116–128, May 1991.

[3] I Tvrdy. From LOTOS to OCCAM. In *Second International Conference on Software Engineering for Real Time Systems*, pages 175–179. The Computing and Control Division of the Institution of Electrical Engineers, September 1989.

[4] A Valenzano, R Sisto, and L Ciminiera. Rapid prototyping of protocols from LOTOS specifications. *Software — Practice and Experience*, 23(1):31–54, January 1993.

[5] H Kopetz, A Damm, C Koza, M Mulazzani, W Swabl, C Senft, and R Zainlinger. Distributed fault-tolerant real-time systems: The MARS approach. *IEEE Micro*, pages 25–40, February 1989.

[6] S Bradley, W Henderson, D Kendall, and A Robson. An Application Oriented Real-Time Algebra. Technical Report NPC-TRS-93-3, Department of Computing, University of Northumbria, UK, 1993.

[7] S Bradley, W Henderson, D Kendall, and A Robson. A formally based hard real-time kernel. Technical Report NPC-TRS-94-3, Department of Computing, University of Northumbria, UK, 1994.

[8] R Alur, C Courcoubetis, and D Dill. Model-checking for real-time systems. In *IEEE Fifth Annual Symposium On Logic In Computer Science*, pages 414–425, June 1990.

[9] J S Ostroff. A verifier for real-time properties. *Real-Time Systems*, 4(1):5–36, March 1992.

[10] C Y Park and A C Shaw. Experiments with a program timing tool based on source-level timing schema. *Computer*, pages 48–57, May 1991.

[11] B Potter, J Sinclair, and D Till. *An Introduction to formal specification and Z.* Prentice-Hall, 1991.

[12] C B Jones. *Systematic software development using VDM.* Prentice-Hall, 1986.

Real-Time Communication in FDDI-Based Reconfigurable Networks *

Wei Zhao, Amit Kumar, Gopal Agrawal,
Sanjay Kamat, Nicholas Malcolm, and Biao Chen
Department of Computer Science
Texas A&M University
College Station, Texas 77843-3112

Abstract

We report our ongoing research in real-time communication with FDDI-based reconfigurable networks. The original FDDI architecture was enhanced in order to improve its fault-tolerance capability while a scheduling methodology, including message assignment, bandwidth allocation, and bandwidth management is developed to support real-time communication. As a result, message deadlines are guaranteed even in the event of network faults.

1 Introduction

Computer networks employed in mission-critical systems must meet stringent timing requirements which arise due to the communication between real-time tasks executing on different network nodes. In this paper, we report our work aimed at addressing issues related to fault-tolerant guarantees of synchronous message deadlines in FDDI-based networks, i.e., the transmission of messages before their deadlines, even in the event of network faults.

FDDI is an ANSI standard [4] for a 100 Mbits/sec fiber optic token ring network. FDDI is a good candidate for mission-critical real-time applications, due not only to its high bandwidth, but also to its property of bounded token rotation time and its dual ring architecture. The bounded token rotation time [20] provides a necessary condition to guarantee hard real-time deadlines, while the dual ring architecture allows the maintenance of continuous real-time service under some failure conditions. Several new civilian and military networks have adopted FDDI as the backbone network. In particular, FDDI has been adopted by the Survivable Adaptable Fiber Optic Embedded Network (SAFENET) [18]. SAFENET is a military standard for computer networks developed with the Navy's Next Generation Computer Resources (NGCR) program.

Although indispensable, the bounded token rotation time and the dual ring architecture alone are inadequate for guaranteeing message deadlines. Several critical issues must be addressed in order to achieve this objective.

- *Fault-tolerance capability must be enhanced.* With the FDDI architecture, only one trunk link fault can be tolerated. Two trunk link faults cause the network to become disconnected. Furthermore, under the current standard, upon the occurrence of a fault, the detection and recovery processes may take several seconds to complete. This is too long to be able to satisfy message deadlines in many hard real-time applications.

- *Message transmission must be properly scheduled.* Message scheduling includes the arbitration of network access for each node, and the control of the length of transmission by each node. The deadlines of messages can be met only if access arbitration and transmission control are performed properly [17].

The above issues are addressed in our project. In particular, we enhance the FDDI architecture in order to improve its fault-tolerance capability. Furthermore, we develop a scheduling methodology to ensure the satisfaction of message time constraints. This work complements previous work on the design of real-time communication networks [1, 6, 7, 8, 9, 12, 13, 14, 19, 21, 23, 24, 25, 26]. For a recent comprehensive survey, the reader is referred to [17].

2 FBRN: An Enhanced FDDI Architecture

We have designed an enhanced network architecture called *FDDI-based reconfigurable network (FBRN)* [11]. An FBRN uses multiple FDDI trunk rings to connect network stations. Specifically, n stations are connected using r FDDI trunk rings. Figure 1(a) shows an FBRN with four FDDI trunk rings connecting four stations. Each station has certain reconfiguration capabilities that provide an additional level of fault-tolerance over that provided by the FDDI *wrap-up*[1] operation. In the case of (multiple) link

*This work was supported in part by AFOSR under Grant F49620-92-J-0385, ONR under Grant N00014-92-J-4031, and an Engineering Excellence Grant from Texas A&M University.

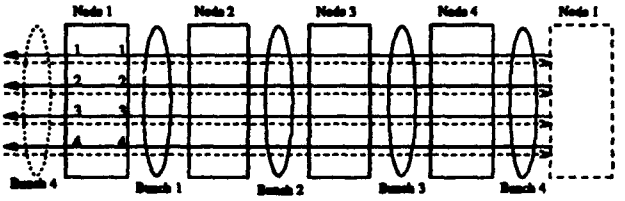
[1] An FDDI trunk ring can recover from a *single* trunk link fault by wrapping up its dual counter-rotating loops [3].

0-8186-5710-3/94 $3.00 © 1994 IEEE

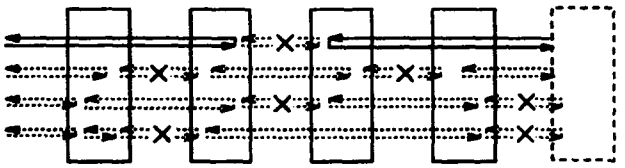
faults, our FBRN automatically detects the occurrence of faults and recovers by reconfiguring the trunk ring connections. This is achieved by judiciously reconnecting the fault-free segments of those trunk rings that could not be recovered by the normal FDDI wrap-up operation. Figure 1(c) demonstrates the reconstruction of such a trunk ring.

Both analytical and simulation data show that our FBRN can sustain a greater number of faults as compared to an ordinary FDDI network. For example, an FBRN consisting of 20 nodes and 4 FDDI trunk rings can still provide, on average, 200 Mbps of transmission bandwidth even if the network has suffered more than 10 link faults.

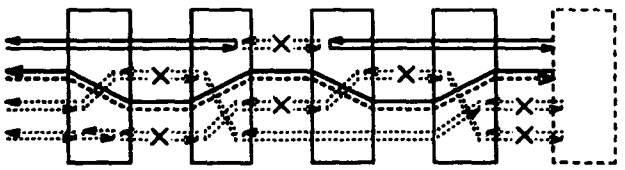
An FBRN can be implemented using existing FDDI concentrator technology. Each node in the FBRN network consists of an enhanced concentrator through which the multiple FDDI trunk rings pass. The concentrator has the capability to reconfigure the connections (paths) among its various ports. A configuration monitor, resident on each FBRN concentrator, uses this ability to route packets among the fault-free ring segments incident on the node. Further details on FBRN implementation are discussed in [11].



(a): An FBRN with n=4, r=4.



(b): Reconfiguration by wrap up in FDDI's link fault management.



(c): Enhanced Reconfiguration by connecting fault-free segments.

Figure 1: FBRN Architecture

3 Scheduling Methodology

We develop a methodology for scheduling message transmissions in an FBRN network so that message time constraints are guaranteed. The methodology consists of the following components:

- *Message assignment method.* To provide deadline guarantees in the presence of trunk link faults, we have to exploit the multi-ring architecture and the fault management mechanism.

  Note that once an FDDI trunk ring is disabled due to faults, messages can only be transmitted on other rings before the faulty ring is recovered. If all rings are fully utilized before the faults occur, it is not possible to transfer message traffic from a faulty ring to a non-faulty ring. Hence, some of the messages will have to be dropped. We assume that when the network is in a faulty state (i.e., some of the FDDI trunk rings are disabled), only a subset of messages that are critical to the mission will be transmitted. The deadlines of critical messages must be guaranteed at all times, including during periods of fault detection and recovery. The objective of this part of the study is to properly assign critical and non-critical messages to FDDI trunk rings so that the deadlines of all messages are met when the network is fault-free, and at least those of critical messages are met when the network is faulty.

  There are three possible methods for achieving this objective:

  - *Fully redundant assignment.* A possible solution is to transmit each critical message over several rings so that when some of the rings are not available, message deadlines are still guaranteed because the messages are transmitted via at least one available ring. This solution is the simplest but it results in wasted bandwidth when there is no fault.
  - *Dynamic reassignment.* Another solution is to dynamically reallocate the messages from one ring to another once a fault is detected. This solution allows better utilization of FDDI rings when there is no fault, but cannot be applied to those applications where the deadlines of critical messages are too small to tolerate any delays caused by fault detection and dynamic reallocation.
  - *An integrated method.* An alternative is to combine the fully redundant method with the dynamic reallocation method. Critical messages with small deadlines are assigned to several rings, and dynamic reassignment is performed for other messages when a link fault occurs on one ring. This method achieves better utilization of the network bandwidth than the fully redundant method, while overcoming the shortcomings of dynamic reallocation.

  We are currently comparing and evaluating the performance of these three approaches in terms of their effectiveness in utilizing the network during both normal and faulty conditions, and in terms of their run-time overhead.

- *Bandwidth allocation method.*

  Once a message is assigned to a particular FDDI trunk ring, its deadline cannot be automatically guaranteed, even though FDDI has a property of 'bounded token rotation time.' Guaranteeing message deadlines is also dependent on the appropriate allocation of the synchronous bandwidth to the nodes. If the source node of a message is allocated insufficient synchronous bandwidth, it may be unable to complete the transmission of real-time messages before their deadline. On the other hand, allocating excess synchronous bandwidths to the nodes could increase the token rotation time, which may also cause message deadlines to be missed.

  Over the past two years, extensive studies have been carried out on this subject. The first comprehensive study on synchronous bandwidth allocation for FDDI was reported in [1]. In [5], an optimal bandwidth allocation algorithm was studied. An optimal algorithm can guarantee deadlines for all messages assigned to an FDDI ring whenever there is an allocation method that can do so. However, the optimal algorithm is complicated and may not be feasible for on-line use. Consequently, in [2], a localized bandwidth allocation method was developed. A localized scheme uses the information local to a node to allocate its synchronous bandwidth. An advantage of localized schemes is that a node can freely change its message parameters and its synchronous bandwidth (as long as the network utilization is within the given bound) without disturbing the operation of other nodes. In [10, 15, 16, 22], extensions were made to the case where messages may have arbitrary deadlines (i.e., the deadlines need not be equal to the periods).

- *Bandwidth management.*

  In the above studies of synchronous bandwidth allocation, it was shown that an FDDI trunk ring where a node may have zero, one, or more streams of synchronous messages can be transformed into a logically equivalent network with one stream per node. This assumption of one stream per node was used to simplify the analysis without loss of generality. However, in practice, having multiple message streams on a node may cause problems. In all the current implementations of the FDDI MAC component, a FIFO queue is used for out-going messages. Furthermore, the FDDI MAC component cannot distinguish between messages from different applications and has no knowledge of the bandwidths allocated to those applications. Whenever the token arrives at a node, the MAC transmits messages in the order of the FIFO queue until it exhausts its total bandwidth.[2] As a result, if messages at the front of the queue are long, messages in the later part of the queue may not be transmitted on time, and hence may miss their deadlines.

  In order to resolve this problem, we need to manage the use of bandwidth at run-time. The particular management functions are: 1) to fragment a message into units of the size of the synchronous bandwidth allocated to the application that generated that message, and 2) to reorder these fragments appropriately before transmission. This reordering is done to arrange the fragments in a sequence equivalent to the transmission sequence that would result if each of these applications were considered to be executing on a separate node. As a result of this fragmentation and reordering, each application is able to transmit a portion of its messages every time the token is received. This ensures that each application is allowed to utilize the synchronous bandwidth allocated to it.

  In our current implementation, the bandwidth management is accomplished by adding a synchronous server (SS), which is a preprocessing module at the application layer. The server receives messages from different applications on the node, fragments the messages, and reorders the fragments as mentioned above, before forwarding them to the device driver.

[2] The synchronous bandwidth allocated to a node is the sum total of the bandwidths allocated to the applications executing on that node.

4 Final Remarks

Finally, we would like to point out that the architecture of our FBRN network is consistent with existing FDDI hardware. Our scheduling method is compatible with the FDDI/SAFENET standard. Hence, the results obtained from our work will be immediately applicable to the design and analysis of distributed hard real-time systems where FDDI/SAFENET networks are used. In fact, SAFENET has adopted our bandwidth allocation method [16], while it is currently considering our FBRN fault management method.

References

[1] G. Agrawal, B. Chen, W. Zhao, and S. Davari, "Guaranteeing Synchronous Message Deadlines with the Timed Token Protocol," *Proc. IEEE International Conference on Distributed Computing Systems*, Yokohama, June 1992.

[2] G. Agrawal, B. Chen, and W. Zhao, "Local Synchronous Capacity Allocation Schemes for Hard Real-Time Communications with the Timed Token Media Access Control Protocol," *Proc. IEEE INFOCOM'93*, 1993.

[3] *FDDI Station Management Protocol (SMT)*, ANSI Standard X3T9.5/84-89, X3T9/92-067, Aug. 6, 1992.

[4] *FDDI Media Access Control (MAC)*, ANSI Standard X3T9.5/88-139, Rev 4.0, Oct 29, 1990.

[5] B. Chen, G. Agrawal, and W. Zhao, "Optimal Synchronous Capacity Allocation for Hard

Real-Time Communications with the Timed Token Media Access Control Protocol," *Proc. IEEE Real-Time Syst. Symp.*, 1992.

[6] D. Ferrari and D. C. Verma, "A Scheme for Real-Time Channel Establishment in Wide-Area Networks," *IEEE Journal on Selected Areas in Communications*, SAC-8:368-379, Apr. 1990.

[7] D. T. Green and D. T. Marlow, "SAFENET – A LAN for Navy Mission Critical Systems," *Proc. Conf. on Local Computer Networks*, Oct. 1989.

[8] R. M. Grow, "A Timed Token Protocol for Local Area Networks," *Proc. Electro/82, Token Access Protocols*, May 1982.

[9] R. Jain, "Performance Analysis of FDDI Token Ring Networks: Effect of Parameters and Guidelines for setting *TTRT*," *IEEE LTS*, May 1991.

[10] S. Kamat, N. Malcolm, and W. Zhao, "The Probability of Guaranteeing Synchronous Real-Time Messages with Arbitrary Deadlines in an FDDI Network," *Proc. IEEE Real-Time Syst. Symp.*, Dec., 1993.

[11] S. Kamat, G. Agarwal and W. Zhao, "On Available Bandwidth in FDDI-Based Reconfigurable Networks," To appear in *Proc. INFOCOM'94*, 1994.

[12] J. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," *Proc. IEEE Real-Time Syst. Symp.*, 1989.

[13] C. C. Lim, L. Yao, and W. Zhao, "A Comparative Study of Three Token Ring Protocols for Real-Time Communications," *Proc. 11th IEEE International Conf. on Distributed Computing Systems*, May 1991.

[14] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM*, 20(1):46–61, Jan. 1973.

[15] N. Malcolm and W. Zhao, "Guaranteeing Synchronous Messages with Arbitrary Deadline Constraints in an FDDI Network," *Proc. 18th IEEE Conf. on Local Computer Networks*, 1993.

[16] N. Malcolm and W. Zhao, "The Timed-Token Protocol for Real-Time Communications," *IEEE Computer*, 27(1):35–41, Jan. 1994.

[17] N. Malcolm and W. Zhao, "Hard Real-Time Communication in Multiple-Access Networks," To appear in *J. Real-Time Systems*.

[18] U.S. Department of Defense, *Survivable Adaptable Fiber Optic Embedded Network*, Jan. 1994. MIL-STD-2204A.

[19] J. Ng and J. Liu, "Performance of Local Area Network Protocols for Hard-Real-Time Applications," *Proc. 11th IEEE International Conf. on Distributed Computing Systems*, May 1991.

[20] K. C. Sevcik and M. J. Johnson, "Cycle Time Properties of the FDDI Token Ring Protocol," *IEEE Trans. Software Eng.*, 13(3), 1987.

[21] L. Sha, and S. S. Sathaye, "A Systematic Approach to Designing Distributed Real-Time Systems," *IEEE Computer*, 26(9):68–78, Sept. 1993.

[22] K. G. Shin and Q. Zheng, "Mixed Time-Constrained and Non-Time-Constrained Communications in Local Area Networks," *IEEE Trans. on Communications*, 44(11):1668–1676, 1992.

[23] J. A. Stankovic and K. Ramamritham, *Hard Real-Time Systems*, IEEE Press, 1988.

[24] J. A. Stankovic and K. Ramamritham, *Advances in Real-Time Systems*, IEEE Press, 1993.

[25] A. M. van Tilborg and G. M. Koob, *Foundations of Real-Time Computing: Formal Specifications and Methods*, Kluwer Academic Publishers, 1991.

[26] A. M. van Tilborg and G. M. Koob, *Foundations of Real-Time Computing: Scheduling and Resource Management*, Kluwer Academic Publishers, 1991.

[27] Q. Zheng and K.G. Shin, "Synchronous Bandwidth Allocation in FDDI Networks," *Proc. ACM First Conference on Multimedia*, 1993.

Session IV: Timing Analysis

Chair: Stuart Faulk
SPC

Correlation Analysis Techniques for Refining Execution Time Estimates of Real-Time Applications *

Rajiv Gupta
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Prabha Gopinath
SSDC Honeywell Inc. MN65-2300
3660 Technology Drive
Minneapolis, MN 55418

Abstract

Scheduling techniques based upon worst case execution times, as are commonly used in real-time applications, often result in severe underutilization of the processor resources since most tasks finish in much less time than their anticipated worst-case execution times. In this paper we describe techniques for identifying correlation among the executions of various statements within a program. We demonstrate how this information can be used to refine the estimate of remaining worst case execution time of a real-time task as the execution of the task progresses. Refined estimates can be used at run-time to achieve better utilization of the system and early failure detection and recovery.

1 Introduction

The success of real-time application software depends on its ability to produce a functionally correct result within definite timing constraints. Hard real-time applications, especially embedded applications, interact with and influence the environment in which they execute. Consequently, safety and timeliness of execution are critical issues since failures have the potential to cause damage. A process in a real-time application has timing-related constraints, such as an Earliest-Start-Time (EST) and a Deadline (DL). It must execute subject to these time constraints and any failure to do so, a deadline failure for instance, constitutes a failure.

The traditional approach to guaranteeing deadlines is to obtain the worst case execution time estimates (WETs) for the individual processes and manually lay out the various execution timelines. While this solution does ensure that the specific set of processes meet their deadlines, there are several obvious problems with such an approach. It does not scale conveniently as the system gets larger[8]. It also often results in severe under-utilization of the system since tasks typically complete in much less time (sometimes orders of magnitude less) than their WETs would indicate [5]. Another problem with the above approach to scheduling based on worst-case execution times arises when a task for some reason, perhaps owing to resource sharing delays, exceeds its WET. This results in deadline failure, but such a failure is noticed very late in the task's lifetime. This reduces the time available to take remedial action. One common solution to this is to add a safety margin by increasing the WET of a task by some arbitrary percentage. This approach further exacerbates the underutilization problem.

In order to address the above problems researchers have proposed run-time refinement of execution time estimates based upon monitoring information [4] [5] [7]. We had introduced a technique called *Compiler Assisted Adaptive Scheduling* (CAADS) [4] using which execution times of the various parts of the program are determined at run-time. If time savings are observed at run-time, they can be used to accommodate newly arriving tasks. On the other hand, if the estimate of the remaining worst case execution time (RWET) of the task is too high, then the decision to abort an executing task is made.

The RWET values will often be much higher than actual remaining execution times. In this paper we present techniques for identifying parts of a program whose executions are correlated with one another. At any given point during the execution of a task, based upon the execution path followed so far and the correlation information, the execution paths that can be followed during the remainder of the task are predicted. This information is then used to estimate the worst case execution time of the remainder of the task, to perform such adaptations as may be required to ensure that deadlines are met, to pre-schedule other related processes, and to pre-allocate resources for this and

*Partially supported by National Science Foundation Presidential Young Investigator Award CCR-9157371 to the University of Pittsburgh.

0-8186-5710-3/94 $3.00 © 1994 IEEE

other processes.

Evidence that a significant degree of correlation exists in programs has been recently provided by Pan *et al* [6]. They found that correlation information significantly improved the accuracy of branch prediction. Their results show that as compared to 2-bit counter-based prediction scheme, the correlation-based branch prediction achieved 11% additional accuracy. In this paper we propose to utilize correlation information for improving RWET estimates. In order to do so we must identify correlation present in a program. Correlation can be detected at compile-time through static analysis techniques and at run-time using profiling techniques. Surprisingly, many of the opportunities for correlation identified by Pan *et al* [6] using run-time history can also be identified through compile-time analysis of a program. However, Pan *et al* [6] did not develop any techniques for identifying correlation through program analysis. In this paper we develop such techniques.

In section 2 we identify the type of correlation information that is useful for refining RWETs and we briefly illustrate how correlation information is used to carry out the refinement. In section 3 we consider an important class of correlations and illustrate the problems that a compiler faces in recognizing these correlations. We briefly outline our approach for compile-time analysis for detecting these correlations. Due to space limitations we will not discuss profiling techniques for correlation detection. Although it is obvious that correlation can be detected through profiling, how to do so efficiently is a challenging problem. Therefore in this respect our efforts are focussed on introducing minimal program instrumentation necessary to collect profile data.

2 Exploiting Correlation Information for Refining RWETs

Intuitively, the correlation between two events, E_1 and E_2, exists if the outcome of E_1 determines the outcome of E_2 under some execution. Thus, correlation between E_1 and E_2 can only exist if after event E_1 has taken place event E_2 can occur. Although it is possible to determine correlation between arbitrary events in a computation, we concentrate on events that effect the execution time of a computation. This approach applies to both sequential and parallel programs. Some examples of events for which correlation values may be useful are as follows:

- The correlation between execution of a statement S (or a set of statements), and the true/false evaluation of a branch B in a sequential program, denoted as $S \rightarrow B^{T/F}$.
- The correlation between execution of a statement S outside a loop L and the number of loop iterations, denoted as $S \rightarrow Iter(L)$.
- The correlation between the call site of procedure P and the true/false evaluation of a branch B in P, denoted as $CallSite(P) \rightarrow B^{T/F}$.
- The correlation between execution of a statement S and creation of a task T in a parallel program, denoted as $S \rightarrow Create(T)$.

We consider the first kind of correlations listed above in the remainder of this paper. However, the broad principles that are used to handle the first kind of correlations are also applicable to other kinds of correlations. The code fragment shown in Figure 1 illustrates the utility of correlation information in estimating RWETs. Each statement is the figure is labeled as `S#(time)`, where time is the execution time of the statement. There is a correlation between the execution of statement `S1` and the false evaluation of the branch `S4` nested inside the for loop, that is, $S1 \rightarrow B0^T$. The RWET immediately preceding the for loop must be considered as 400 time units if no correlation information is available since in this situation we must assume that `S4` is executed during each iteration of the loop. However, once having recognized that there is a correlation between the execution of `S1` and the false evaluation of `B0`, we can obtain a better RWET estimate. By recognizing that `S1` has been executed at run-time, we can consider the RWET preceding the for loop to be 300 time units instead of 400 time units.

```
S0(1): if (..) { S1(1):  a = 0 }
S2(1): for ( i = 1; i != 101; i++ ) {
                    S3(1):  ...
                    B0(1): if (a!=0) {
                                S4(1):  ...
                           }
       }
```

Figure 1. Estimating RWET using Correlation.

The above example illustrates a common situation that Pan *et al* [6] observed in many SPEC integer benchmark programs such as the `gnu C-compiler`, `eqntott`, `li` etc. Very often there are statements that assign constant values to variables that are typically flags. Later in the program the flag is checked to determine whether or not a body of code should be executed.

3 Computing Correlation Information

In this section we describe various situations in which there is a correlation between the execution of an assignment statement and the true/false evaluation of a branch encountered later in the program. With each different situation the compiler is faced with different challenges. The solutions to these challenges are integrated to develop an algorithm which is also presented in this section.

The basic approach taken by the compiler is to first identify the conditional branches whose outcome significantly affects the program execution time. Next in order to predict each such branch the compiler examines the assignment statements that directly or indirectly affect the values of variables referenced by the branch condition. In other words, given a branch B, we must search for a statement S such that $S \rightarrow B^{T/F}$ exists.

1. **Directly Affecting Constant or Non-constant Assignment:** If S assigns a value to a variable which is compared with a constant or another variable to determine the outcome of B, and the outcome can be determined at compile-time then we have identified correlation $S \rightarrow B^{T/F}$. The example in Figure 1 illustrated this situation. Application of symbolic evaluation can expose correlations that are not as readily observable as the case in Figure 1. For example in Figure 2a the correlation $S2 \rightarrow B0^{T}$ requires symbolic evaluation of the branch condition.

```
S0:  ...
S1:  if (..) { S2:  y = a + 1 }
B0: if (y != a) { S4:  .... }
```

Figure 2a. Correlation Detection Requiring Symbolic Evaluation.

```
B0:  while (not done) {
          S1:  ..
          S2:  if (..) { S3:  done = true }
          S4:  ..
}
```

Figure 2b. Correlation Indicating Loop Termination.

In example shown in Figure 2b the correlation $S3 \rightarrow B0^{F}$ can be detected. The correlation essentially identifies the loop termination condition. The worst case execution time of the while loop can be computed accurately using this information. To compute the worst case time we should assume that the if-condition in S2 evaluates to false in all iterations but the last. If correlation information was not available, we would have assumed true evaluation for the if-condition during all loop iterations for computing the worst case execution time.

2. **Directly Affecting Multiple Assignments:** In some situations the branch condition may reference multiple variables and hence correlation relationship may involve more than one assignment statement. The example in Figure 3 taken from SPEC benchmark **eqntott** illustrates this situation. The branch $S4$ evaluates to false if statements $S1$ and $S3$ are executed, that is, $S1 \wedge S3 \rightarrow Exec(B0^{F})$. In this situation the compiler must find all definitions of each variable used in a branch that reach the branch. Next a combination of assignment statements corresponding to the variables are selected such that the corresponding values of variables enable the evaluation of the branch and the execution of these statements is not mutually exclusive. Each such selection provides us with a correlation that allows the prediction of branch outcome under certain conditions.

```
S0: if (aa==2) { S1:  aa = 0 }
S2: if (bb==2) { S3:  bb = 0 }
B0: if (aa!=bb) { S5:  .... }
```

Figure 3. Correlation Involving Multiple Assignment Statements.

3. **Indirectly Affecting Assignments:** In some situations in order to identify correlations we need to compute a program data slice [9] which identifies all those statements that *directly* or *indirectly* influence the values of variables used in a branch condition. Consider the example shown in Figure 4. The program slice for the value of y in $S5$ is includes statements $S1$, $S2$ and $S4$. By systematically searching this slice we can determine that the execution of $S1$ followed by the execution of $S4$ will cause branch $S5$ to evaluate to true, that is, $S1 \wedge S4 \rightarrow B0^{T}$. As we can see this situation can't be handled by earlier techniques discussed in this section.

```
S0: if (..) { S1:  x = 1 }
    else { S2:  x = a }
S3: if (..) { S4:  y = x }
B0: if (y==1) { S5:  .... }
```

Figure 4. Indirect Correlation Requiring Program Slicing.

The above example illustrates that in order to detect all correlations we must consider both statements that directly influence the branch variables and those that indirectly influence them. By combining the ideas of symbolic evaluation, multiple assignments and direct/indirect influences we develop a general algorithm for identifying correlations. The main steps of the algorithm are as follows:

1. A *control flow graph* (CFG) representation of the program is constructed [1].

2. The program is converted to *static single assignment* (SSA) form [2]. The conversion of a program into this form guarantees that each variable is reachable from a single definition of that variable, that is, the definition-use relationships in the program are explicit in the program. This representation simplifies the algorithms used in future steps. To achieve this goal a ϕ function corresponding to a variable is introduced at a join point if different definitions of the variable reach the join point along different paths. The ϕ function indicates the selection of the appropriate value of the variable based upon the path along which the control reaches the join point.

3. Using some simple heuristics, the branch conditions whose results are likely to affect the execution time of a program significantly are identified.

4. For each of branch condition, B, identified in the preceding step, we identify the statements that influence its outcome as follows.

- For each variable v, used in branch B, we compute the corresponding *data slice*, $DS(v)$. The data slice contains all statements that directly or indirectly lead to the computation of the value of v used in the branch condition. Thus, the data slice is computed by taking a closure of data dependences for the variable [9].

- Corresponding to each variable v, using the data slice $DS(v)$ and forward substituting expressions, we generate a set of symbolic expressions, $SE(v)$, that represent the value of variable v at B under various program executions. The set of control conditions that must hold for a given expression $e \in SE(v)$ is denoted as $CC(v, e)$. Depending upon the complexity of SE and CC sets that one is willing to allow, we can devise algorithms for computing SE and CC sets with varying degree of complexity. We intend to include only those expressions in SE for which the corresponding control condition in CC includes a single predicate. The reason for this restriction is that the run-time overhead associated with capturing control conditions at run-time can be limited through this assumption. Special consideration of loops is required during the generation of SE sets. Expressions from loops are only forward substituted if they represent invariant computations. This condition ensures that the presence of loops does not generate unbounded number of expressions in the SE sets.

- For each combination of expressions for the branch variables, taken from their respective SE sets, we attempt an evaluation of the branch condition B using symbolic analysis. Based upon these evaluations we prepare a set of combinations $EVAL(B)$ for which the branch was successfully evaluated using symbolic analysis.

- For each possible combination of variable expressions in $EVAL(B)$, identified in the preceding step, we check the control conditions (CCs) under which the expressions hold to ensure that the branch variables can simultaneously hold these values in some program execution. If the control conditions can hold simultaneously, we have identified a possible correlation under which the outcome of the branch condition B can be predicted.

The above discussion provided the main steps for correlation detection. The details of the various steps are omitted due to space limitations. In Figure 5 we present an example to briefly illustrate the above algorithm. The SSA for of the code fragment in Figure 5a is given in Figure 5b. Due to the renaming of variables and the introduction of ϕ functions, the data flow relationships have been made explicit. The SE and CC sets for the three branches are given in Figure 5c. Using the SE values we evaluate the branches and get the true/false evaluations of branches in the instances shown in Figure 5d. After checking the feasibility of these evaluations we detect the correlations listed in Figure 5e.

```
S0:  read y
S1:  inc = 1
B0: if (y > 0) { S2:  z = 1;
                 S3:  w = 2; S4:  x = y - inc }
    else { S5:z=2; S6:w=1; S7:  x=y+inc }
B1:  if (x < y) { ... }
B2:  if (w = z) { ... }
```

Figure 5a. Example Program.

```
S0:  read y
S1:  inc = 1
B0: if (y > 0) { S2:  z1=1;
                 S3:  w1=2; S4:  x1=y-inc }
    else { S5:z2=2; S6:w2=1; S7:x2=y+inc }
    z3 = φ(z1, z2)
    w3 = φ(w1, w2)
    x3 = φ(x1, x2)
B1:  if (x3 < y) { ... }
B2:  if (w3 = z3) { ... }
```

Figure 5b. SSA form of the Program.

```
B0:
    SE(y) = {y}
B1:
    SE(x3) = {y-1,y+1};
    CC(x3,y-1) = { y>0 }; CC(x3,y+1) = { ¬y>0 }
    SE(y) = {y}; CC(y,y) = {true}
B2:
    SE(w3) = {2,1};
    CC(w3,2) = { y>0 }; CC(w3,1)= { ¬y>0 }
    SE(z3) = {1,2};
    CC(z3,1) = { y>0 }; CC(z3,2) = { ¬y>0 }
```

Figure 5c. The SE and CC Sets.

```
B0:  no evaluation achieved.
B1:
    x3=y-1 ⇒ B1^T
    x3=y+1 ⇒ B1^F
B2:
    (w3,z3)=(2,1) ⇒ B2^F; (w3,z3)=(2,2) ⇒ B2^T
    (w3,z3)=(1,1) ⇒ B2^T; (w3,z3)=(1,2) ⇒ B2^F
```

Figure 5d. Branch Evaluation.

```
B0:  no correlation detected.
B1:  S4 → B1^T; S7 → B1^F
B2:  S2/S3 → B2^F; S5/S6 → B2^F
```

Figure 5e. Correlation Detected.

```
S0:  y = 1
B0: while (..) {
                B1:  if (..) { S1:  x = y + 1 }
                S2:  ..
    }
B2:  if (x < 3) { .. }
```

Figure 6. Forward Substitution in Loops.

The example in Figure 6 illustrates the detection of correlation in presence of a loop. If statement $S1$ is loop invariant and there are no other definitions of x in the loop, then we generate an expression for the value of x computed at $S1$. This results in the detection of correlation $S1 \rightarrow B2^T$. In all other situations the generation of expressions through the loop would have been discontinued and no correlation for $B2$ would have been detected.

4 Concluding Remarks

In this paper we have introduced compiler techniques for detecting and exploiting correlation information for refining RWETs. We did not consider parallel programs in this paper. However, methods for statically analyzing distributed programs including slicing algorithms also exist [3]. Thus we believe that with further research our approach can be extended to handle distributed programs.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently Computing Static Single Assignment Form and Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 4, pages 451-490, October 1991.

[3] E. Duesterwald, R. Gupta and M.L. Soffa. Distributed Slicing and Partial Re-execution for Distributed Programs. *Proc. Fifth Workshop on Languages and Compilers for Parallel Computing*, LNCS 757 Springer Verlag, pages 497-511, August 1992.

[4] P. Gopinath and R. Gupta. Applying Compiler Techniques to Scheduling in Real time Systems. *Proc. 11th Real-Time Systems Symposium*, pages 247-256, Orlando, Florida, December 1990.

[5] D. Haban and K.G. Shin. Application of Real-Time Monitoring to Scheduling Tasks with Random Execution Times. *Proc. 10th Real-Time Systems Symposium*, 1989.

[6] S-T. Pan, K. So and J.T. Rahmeh. Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. *Proc. Architectural Support for Programming Languages and Operating Systems*, pages 76-84, 1992.

[7] C. Park and A. Shaw. Experiments with a Program Timing Tool Based on Source-Level Timing Schema. *Proc. 11th Real-Time Systems Symposium*, pages 72-81, 1990.

[8] L. Sha and J. Goodenough. Real Time Scheduling Theory and ADA. *Computer*, pages 53-62, April 1990.

[9] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, pages 352-357, July 1984.

Issues of Advanced Architectural Features in the Design of a Timing Tool*

Byung-Do Rhee Sung-Soo Lim
Sang Lyul Min Heonshik Shin
Chong Sang Kim

Dept. of Computer Engineering
Seoul National University
Seoul 151-742, Korea

Chang Yun Park

Dept. of Computer Engineering
Chung-Ang University
Seoul 156-756, Korea

Abstract

This paper describes a timing tool being developed by a real-time research group at Seoul National University. Our focus is on the issues resulting from advanced architectural features such as pipelined execution and cache memories found in many modern RISC-style processors. For each architectural feature we state the issues and explain our approach.

1 Introduction

In real-time computing systems, tasks have timing requirements (i.e., deadlines) that must be met for correct operation. Various scheduling techniques have been proposed to guarantee such timing requirements. In many cases, these scheduling techniques require that the worst case execution times (WCETs) of tasks be known a priori.

This paper describes a timing tool that is being developed by a real-time research group at Seoul National University. This timing tool aims at accurately calculating guaranteed worst case execution times of programs for computer systems that use modern RISC-style microprocessors. Our particular focus is on how the timing tool addresses the issues resulting from advanced features of these microprocessors such as pipelined execution and cache memory. There have been various approaches to predicting program execution times [2, 5, 8, 9, 11, 12]. However, their machine models were mostly CISC-style processors rather than RISC-style microprocessors.

*This work was supported in part by ADD (Contract ADD-91-4-4) and KOSEF (Grant KOSEF-93-01-00-10).

This paper is organized as follows. Section 2 presents the overview of the timing tool. In section 3, we explain the problems in accurately estimating the WCETs of tasks in pipelined processors and present an analysis method based on extended timing schema. Section 4 explains why an accurate timing analysis is difficult in computer systems with cache memories and briefly discusses our approach. Finally, we conclude this paper in section 5.

2 Overview of the timing tool

Our timing tool, like [7, 8], is based on the timing schema [10]. The timing schema is a set of formulas for computing the time-bounds of programming constructs. For example, the time-bound of **S: if (exp) then S_1; else S_2** is computed by the following equations:

$$\begin{aligned} T(S) &= T(S_{then}) \biguplus T(S_{else}) \\ T(S_{then}) &= T(exp) + T(then) + T(S_1) \\ T(S_{else}) &= T(exp) + T(else) + T(S_2) \end{aligned}$$

where $T(exp)$, $T(S_1)$ and $T(S_2)$ are the time-bounds of exp, S_1, and S_2, respectively and $T(then)$ and $T(else)$ are the time-bounds to transfer control to S_1 and S_2, respectively. The operation $\biguplus$ on time-bounds is defined as

$$[a, b] \biguplus [c, d] \equiv [min(a, c), max(b, d)]$$

Our timing tool consists of a compiler and a timing analyzer. The compiler is a modified version of

0-8186-5710-3/94 $3.00 © 1994 IEEE

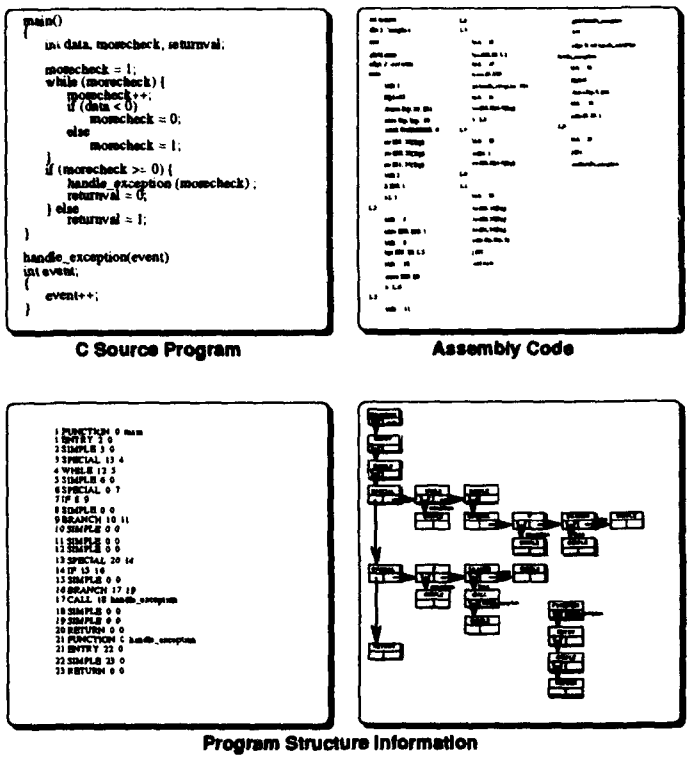


Figure 1: Sample C source program and the output from the compiler

an ANSI compiler called lcc [1]. This compiler accepts a C source program and generates the assembly code along with program structure information. Figure 1 shows a sample C program and the generated assembly code. Also shown in the figure is the program structure information in both textual and graphical forms. The timing analyzer uses the assembly code and the program structure information along with user-provided information (e.g., iteration counts of loop statements, WCETs of the library functions used in the program) to compute the time-bound of the program. The machine model currently supported in the timing tool is the MIPS R3000 CPU.

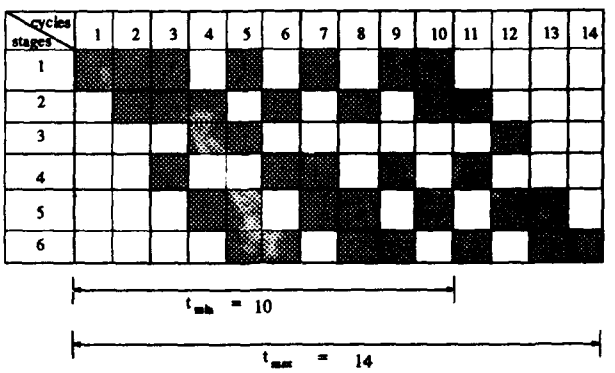


Figure 2: Typical reservation table

3 Pipelining effects

Due to data dependencies and resource conflicts within the execution pipeline, the execution time of a basic block will differ depending on which basic block among the possible basic blocks was executed prior to this basic block. In the original timing schema, it is difficult to accurately account for such timing variations since the basic object of the timing schema is a simple time-bound. To rectify this problem, we extended the original timing schema. In the extended timing schema, the basic object is a reservation table rather than a simple time-bound. The reservation table was originally proposed to describe and analyze activities within a pipeline [3] (cf. Figure 2). In the reservation table, the vertical dimension represents the stages in the pipeline and the horizontal dimension represents time. The shaded boxes in the reservation table specify the use of the corresponding stages for the indicated period of time. In our approach, reser-

vation tables are used to specify timing of instruction executions. In our timing tool, associated with each reservation table are its worst and best case execution times that are denoted by t_{max} and t_{min} respectively in Figure 2.

The use of reservation tables as the basic objects of the timing schema allows us to rewrite the timing schema in such a way that takes into account the timing variation due to dependencies between programming constructs. For example, in the extended timing schema, the timing schema of **S: S_1; S_2** is

$$R(S) = \{r | r = r_1 \oplus r_2,\ r_1 \in R(S_1),\ r_2 \in R(S_2)\}$$

where $\oplus$ is an operation that concatenates two reservation tables giving another reservation table. $R(S_1)$ is the set of reservation tables corresponding to the set of the execution paths that *might* take the longest time among the possible execution paths in S_1. $R(S_2)$ is defined similarly. During each instantiation of the above timing schema, a check is made to see whether the resulting set of reservation tables can be pruned. A reservation table can be pruned if the worst case execution time of the reservation table is shorter than the best case execution time of another reservation table in the same set. Note that in such a case the execution path corresponding to a pruned reservation table cannot be the worst case execution path. Figure 3 shows an example of such pruning.

Likewise, the timing schema of an **if** statement **S: if (exp) then S_1; else S_2** is

$$\begin{aligned} R(S) \quad = \quad & \{r_a | r_a = r_{exp} \oplus r_1,\ r_{exp} \in R(exp),\ r_1 \in R(S_1)\} \\ & \bigcup \\ & \{r_b | r_b = r_{exp} \oplus r_2,\ r_{exp} \in R(exp),\ r_2 \in R(S_2)\} \end{aligned}$$

where $\bigcup$ is the set union operation. As in the previous example, pruning is performed each time a new set of reservation tables is derived.

In the current implementation, not all the columns in the reservation table are stored. Instead only a first few columns whose timing behavior may be affected by the preceding basic block and a last few columns who may affect the timing behavior of the succeeding basic block are maintained.

4 Cache Memories

Our timing tool currently does not support cache memories that have been extensively used to bridge the speed gap between the processor and main memory. In a cache-based computer system, we need to know the cache hit or miss of each memory reference to locate the worst case execution path. Unfortunately such information is known only after the worst case execution path has been found due to history-sensitive nature of caches. This cyclic dependency, in many cases, yields a pessimistic estimation of WCETs [6]. To rectify the problem resulting from the cyclic dependency, we again extended the timing schema [4]. We will briefly describe the technique in the following.

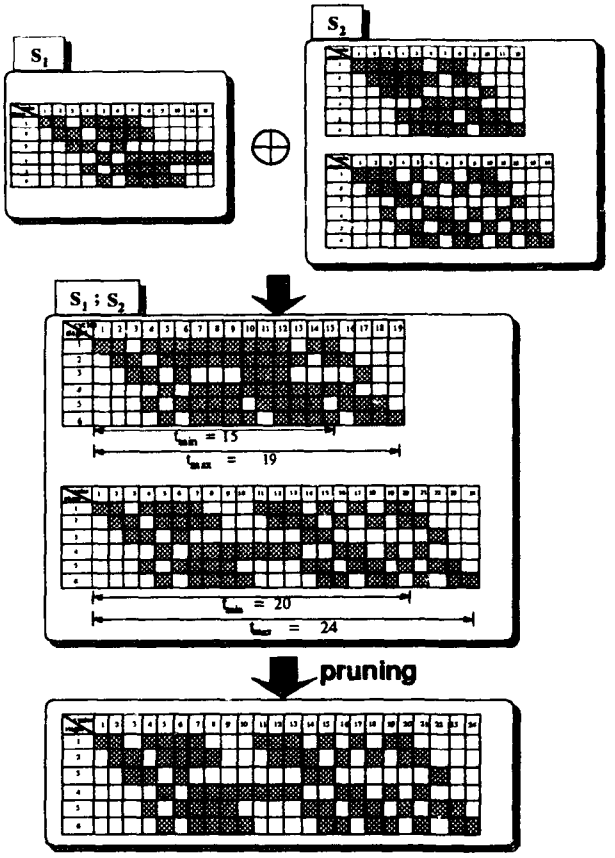


Figure 3: Example of pruning

In the proposed technique, associated with each statement is a set of atomic objects, each of which abstracts an execution path in the statement. Each atomic object consists of two sets of memory block addresses and an execution time estimate. The first set maintains the memory block addresses of the references whose hits or misses depend on the cache contents before the statement. In other words, this set maintains for each cache block the memory block address of the first reference to the cache block. The second set maintains the addresses of the memory blocks that will remain in the cache after the execution of

the statement. In other words, this set maintains for each cache block the memory block address of the last reference to the cache block. This is the cache contents that will determine the hits or misses of memory references from succeeding statements. The execution time estimate is an estimated time needed to execute the statement. In this estimate, correctly accounted for are the "guaranteed" hits and misses. However, the memory references whose hits or misses are not known (i.e., those in the first set) are conservatively assumed to miss in the cache in the initial estimate. This initial estimate is later refined as the hits or misses of those references are known in a later stage of analysis. This framework allows us to rewrite the timing schema so as to accurately analyze the timing behavior of cache memories. A detailed discussion of the resultant timing schema is beyond the scope of this paper and interested readers are referred to [4]. However, it is worth mentioning that the resulting timing schema is very similar to the one given in the previous section and that the two timing schemas can easily be combined.

5 Conclusion

In this paper, we explained the difficulty of accurately estimating the time-bounds of programs in RISC-based computer systems by using as an example the timing tool we are currently developing. Our particular focus was on pipelined execution and cache memories that are typical of RISC-based computer systems.

On the pipelined execution, we explained the limitation of the original timing schema and described an extension of it that is based on reservation tables. On the memory hierarchy, we explained why an accurate timing analysis is difficult in computer systems with cache memories and described our approach.

We expect that an accurate analysis of combined effects of pipelined execution and cache memory on program execution time will be possible when the planned extension of our timing tool is completed. This will allow RISC-style processors to be widely used in real-time systems without worrying about their unpredictable worst case performance.

References

[1] C. W. Fraser and D. R. Hanson. A code generation interface for ANSI C. Technical Report CSL-TR-270-90, Princeton University, July 1990.

[2] M. Harmon, T. P. Baker, and D. B. Whalley. A retargetable technique for predicting execution time. In *Proceedings of the 13th Real-Time Systems Symposium*, pages 68–77, 1992.

[3] P. M. Kogge. *The Architecture of Pipelined Computers*. Hemisphere Publishing Corp., 1981.

[4] S.-S. Lim, S. L. Min, C. Y. Park, K. Park, Heonshik Shin, and C. S. Kim. An accurate instruction cache analysis technique for real-time systems. To appear in the Workshop on Architectures for Real-time Applications, 1994.

[5] A. Mok. Evaluating tight execution time bounds of programs by annotations. In *Proceedings of the 6th IEEE Workshop on Real-Time Operating Systems and Software*, pages 74–80, 1989.

[6] F. Mueller, D. Whalley, and M. Harmon. Predicting instruction cache behavior. Unpublished Technical Report, 1993.

[7] C. Y. Park. *Predicting Deterministic Execution Times of Real-Time Programs*. PhD thesis, University of Washington, August 1992.

[8] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 72–81, 1990.

[9] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems*, 1(2):159–176, Sept. 1989.

[10] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions On Software Engineering*, 15(7):875–889, July 1989.

[11] A. Stoyenko. *A Real-Time Language with a Schedulability Analyzer*. PhD thesis, University of Toronto, Dec. 1987.

[12] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst-case execution times. *Journal of Real-Time Systems*, 5(4):319–343, Oct. 1993.

Timing Analysis of Superscalar Processor Programs Using ACSR

Jin-Young Choi, Insup Lee, and Inhye Kang

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389

Abstract

This paper illustrates a formal technique for describing the timing properties and resource constraints of pipelined superscalar processor instructions at high level. Superscalar processors can issue and execute multiple instructions simultaneously. The degree of parallelism depends on the multiplicity of hardware functional units as well as data dependencies among instructions. Thus, the timing properties of a superscalar program is difficult to analyze and predict.

We describe how to model the instruction-level architecture of a superscalar processor using ACSR and how to derive the temporal behavior of an assembly program using the ACSR laws. The salient aspect of ACSR is that the notions of time, resources and priorities are supported directly in the algebra. Our approach is to model superscalar processor registers as ACSR resources, instructions as ACSR processes, and use ACSR priorities to achieve maximum possible instruction-level parallelism.

1 Introduction

Instruction-level parallelism is widely used in superscalar processors to improve execution speed. Superscalar processors realize instruction-level parallelism by replicating functional hardware and by overlapping instruction execution stages in pipeline [7, 5, 8]. Consequently, multiple instructions can be issued and executed simultaneously in superscalar processors. The degree of parallelism depends on the multiplicity of hardware functional units as well as data dependencies among instructions. One of difficulties in using superscalar processors for time critical applications is that it is difficult to predict the timing behavior of programs.

Our goal is to augment the Instruction Set Architecture (ISA) level [3] description with timing properties and resource constraints using a formal technique based on process algebra. We use ACSR, Algebra of Communicating Share Resources, because it includes the notions of time, resources and priorities. There are several advantages for using ACSR. First, the notion of ACSR resource facilitates the modeling of processors and instructions. A superscalar processor can be defined as a set of reusable resources such as registers. Then, an instruction can be represented by a process which acquires and consumes a subset of resources in time. Second, the concept of maximal resource utilization which is a fundamental idea for pipeline and superscalar processors, can be described. This is done using the notion of ACSR priority which allows the specification of scheduling among several possible alternatives. Third, ACSR has a proof technique that can be used to verify properties of instruction specifications written in ACSR. There also is an available tool, called VERSA [2], which allows the programmer to interactively execute, analyze, and rewrite ACSR specifications.

Our formal processor specification based on ACSR is useful in several areas. For instance, the specification provides an assembly programmer precise meaning of instructions with respect to timing behavior and resource use. This information is essential when a programmer wants to use the processor for time critical systems. Furthermore, the specification can serve as documentation between instruction set designer and hardware implementor. Since the ACSR interpreter already exists, the development of a timing analyzer for a new superscalar processor requires only a simple translator from instructions to ACSR processes. Finally, a translated superscalar program helps one to analyze a sequence of instructions in terms of data dependencies. Such data flow information is useful in code generation and optimization for compilers.

Our work was inspired by the pioneering work by Harcourt *et al.* which uses a process algebra called SCCS for instruction specification [4]. Our approach

0-8186-5710-3/94 $3.00

differs from theirs as follows. Since SCCS does not have the notion of resources, they represent each resource using a binary semaphore process. The resulting specification becomes quite complicated and cumbersome. Our specification is clear and natural, because of the explicit notion of resources in the algebra. Furthermore, since SCCS does not have priority concept, they adapted a priority operator developed for CCS [1] in the context of SCCS. On the other hand, ACSR has the built-in notion of priority, which can be used to model the maximum parallel and pipeline execution of instructions.

To illustrate our approach, this paper uses a hypothetical superscalar processor, called ToyP, developed by Harcourt *et al.* [4]. The ToyP processor includes many features of commercial processors, such as delayed loads and branches, interlocked floating-point instructions, and multiple instruction issue. To simplify our presentation, we model only integer instructions such as add, move, load, store instructions. We assume that add and move instructions perform its task in a single instruction cycle, whereas memory-related instructions, load and store, need two instruction cycles.

The rest of the paper is organized as follows. In Section 2, we introduce a hypothetical ToyP superscalar processor and present a subset of ACSR and review some basic properties. Section 3 describes our approach for specifying instructions by using ACSR. In Section 4, we demonstrate how a ToyP program can be translated into ACSR and its execution simulated. Section 5 summarizes the paper and describes plans for future work.

2 ACSR for ToyP Processor

This section introduces the ToyP superscalar processor and our basic formalism ACSR, Algebra of Communicating Shared Resource.

ToyP: a Simple 32-bit Processor. ToyP was designed by Harcourt *et al.* to illustrate the specification of instruction-level parallelism [4]. ToyP is a simple hypothetical RISC with 32-bit instructions, memory word size, registers and addresses. To simplify our presentation, we only consider a subset of ToyP instructions: **add, mov, load, store.**

add	Ri,	Rj,	Rk	Ri ← Rj + Rk
mov	Ri,	Rj		Ri ← Rj
load	Ri,	Rj,	#c	Ri ← Mem[Rj + c]
store	Ri,	Rj,	#c	Mem[Rj + c] ← Ri

The instructions **add** and **mov** perform its task in a single instruction cycle, while memory-related instructions, **load** and **store**, need two instruction cycles. Thus, when a **load** or **store** instruction is executed, the result is available after 2 instruction cycles.

Algebra of Communicating Shared Resource. ACSR is a real-time process algebra that incorporates the notions of communication, concurrency, resources, and priorities into a single formalism. One of the important concepts in ACSR is shared resources. We briefly describe the subset of ACSR which we use to model microprocessor instructions; the detailed description and semantics of ACSR can be found in [6].

We consider a system to be composed of a finite set R of registers with priority 1 in ToyP processor. An action is defined as a subset of R, that consumes one cycle of time. As an example, the singleton action, $\{(r,1)\}$, denotes the use of some register $r \in R$. For the simplicity, we omit the priority from actions hereafter. The action $\emptyset$ represents idling for one time unit, since no resource is being used. We let A, B, C range over actions.

The syntax of ACSR processes containing actions is as follows:

$$P ::= \text{NIL} \mid A : P \mid P_1 + P_2 \mid [P]_I \mid P_1 \| P_2 \mid rec\ X.P \mid X$$

NIL is a process that executes no action (i.e., it is deadlocked). There is one prefix operator. $A : P$ executes a resource-consuming action A, consumes one time unit, and proceeds to the process P. The Choice operator $P_1 + P_2$ represents nondeterminism – either of the processes may be chosen to execute, subject to the resource limitations of the environment. The operator $P_1 \| P_2$ is the concurrent execution of P_1 and P_2. The Close operator, $[P]_I$, produces a process P that monopolizes the resources in $I \subseteq \mathcal{R}$. The process $rec\ X.P$ denotes standard recursion, allowing the specification of infinite behavior.

We denote Done as $rec\ X.\emptyset : X$. The process Done has the following identity property with respect to the parallel composition operator $\|$: For every process P, $P \| \text{Done} = P$. We introduce one binary combinator *next* which is used to model the issuing of instructions in consecutive cycles.

Definition 2.1 *For any A, P, Q, the next operator is defined as follows:*

$$(A : P)\ next\ Q \stackrel{\text{def}}{=} A : (P \| Q).$$

add Ri, Rj, Rk	$\stackrel{def}{=}$	$\sum_{1\le l\le 6}\sum_{1\le m\le 6}\{$ Ri, Rj_l, $Rk_m\}$: Done
mov Ri, Rj	$\stackrel{def}{=}$	$\sum_{1\le l\le 6}\{$ Ri, $Rj_l\}$: Done
load Ri, Rj, #c	$\stackrel{def}{=}$	$\sum_{1\le l\le 6}\{$ Ri, $Rj_l\}$: {Ri} :Done
store Ri, Rj, #c	$\stackrel{def}{=}$	$\sum_{1\le l\le 6}\sum_{1\le m\le 6}\{$ Ri_l, Rj_m }: $\{Ri_l\}$: Done

Table 1: **Modeling ToyP Instructions Using ACSR**

3 Modeling ToyP Instructions Using ACSR

We model a ToyP instruction as an ACSR process and hardware components needed for execution of instructions as resources in ACSR. For this paper, we need to consider only integer registers as resources. An instruction is modeled by an ACSR process which specifies the behavior of the instruction in terms of resource constraints and temporal properties, that is, which and when resources are needed.

There are two kinds of operations on an integer register: read and write. A register can be shared by the executions of several instructions if all of them are reading a value from the register. However, only one instruction can use a register if an instruction is trying to write a value into the register. If an instruction needs a register whose use is not compatible with a currently executing instruction, then the execution of the new instruction must be delayed.

Since ACSR resources are serially reusable, we represent each register as consisting of multiple ports for read and write. The number of ports depends on how many instructions can be issued and executed in parallel. We assume that ToyP can issue and execute up to three integer instructions in a single cycle if there are no data hazard. Since each integer instruction can read a single register twice (e.g., in **add** R2, R1, R1), each register can be accessed by 6 read requests in the same cycle. Instead of modeling registers themselves, we model these 6 read/write (serially reusable) ports for each register. Here, a process can read a value if it acquires any one of six ports, whereas a process can write a value only if the process acquires all six ports of the register. Hence, when a process holds a register for writing, other processes cannot share the register. Ri_j denotes the port j of the register i. For the sake of simplicity, we sometimes use Ri to mean all 6 ports of the register i, that is, R: $\stackrel{def}{=}$ $\{Ri_1, Ri_2, Ri_3, Ri_4, Ri_5, Ri_6\}$. Hence, $\{Rj_1, Ri\}$ stands for $\{Rj_1, Ri_1, Ri_2, Ri_3, Ri_4, Ri_5, Ri_6\}$.

We model the ToyP instructions using ACSR as in table 1.

To simplify the presentation, we choose one alternative from the several choices for the translation of each instruction. For example, we translate **add** R1, R2, R3 into $\{R1, R2_1, R3_1\}$:Done instead of $\sum_{1\le l\le 6}\sum_{1\le m\le 6}\{$ R1, $R2_l$, $R3_m\}$: Done.

Using the above translation scheme for ToyP instructions, a ToyP program can be represented by a set of ACSR processes. The resulting set of ACSR processes is called a program specification. The next example illustrates how to translate a ToyP programs to a program specification.

Example 3.1 The following program is assumed to be loaded into the memory starting at the location *PC*:

PC :	**add** R1, R1, R1
PC+4 :	**load** R2, R3, #8
PC+8 :	**add** R1, R3, R3

The above program sequence can be represented by the following ACSR processes:

Mem(PC)	$\stackrel{def}{=}$	{ R1 } : Done
Mem(PC+4)	$\stackrel{def}{=}$	{ R2, $R3_1$ } : {R2} : Done
Mem(PC+8)	$\stackrel{def}{=}$	{ R1, $R3_2$, $R3_3$} : Done
Mem(PC+12)	$\stackrel{def}{=}$	NIL

4 Modeling Superscalar ToyP execution

This section describes how to model the superscalar aspect of ToyP that can issue multiple instructions per cycle using ACSR. The ToyP processor can issue and execute multiple (up to 3) instructions at the same time. In the superscalar architecture, hardware determines register use conflict (called data hazard) among instructions. A ToyP program contains no explicit synchronization information to prevent data hazard. Hence, in order to predict the timing behavior of a ToyP program, it is necessary to be able to determine when there are data hazard.

$$
\begin{aligned}
\text{Insts(PC)} &\stackrel{\text{def}}{=} \emptyset : \text{Inst(PC)} + \text{Super-Insts(PC)} \\
\text{Super-Insts(PC)} &\stackrel{\text{def}}{=} (\text{Mem(PC)} \parallel \text{Mem(PC+4)} \parallel \text{Mem(PC+8)}) \\
&\qquad \text{next Insts(PC+12)} \\
&\quad + (\text{Mem(PC)} \parallel \text{Mem(PC+4)}) \text{ next Insts(PC+8)} \\
&\quad + \text{Mem(PC) next Insts(PC+4)} \\
\text{Program} &\stackrel{\text{def}}{=} [\text{Super-Insts(PC)}]_R
\end{aligned}
$$

Figure 1: Modeling the Execution of ToyP Processor

Data Hazard. A value generated by execution of an instruction is used by other instructions, and such a flow of data during program execution creates data dependencies. A sequence of instructions with data hazard cannot be executed in a different order or concurrently. Hence, they must be executed in the given order in the instruction sequence. Example 3.1 contains data hazard on register R1 since it is write accessed by the first instruction and read-accessed by the third instruction. Data hazard such as read-after-write, write-after-read, write-after-write hazard is presented as a resource conflict in ACSR, hence NIL.

The semantics of ACSR prevents concurrent executions of instructions with potential data hazard as can be seen from the following lemma.

Lemma 4.1 *Given two ACSR processes, $A : P$ and $B : Q$, that represent two integer instructions, $A : P \| B : Q$ = NIL iff $A : P$ and $B : Q$ have a data hazard.*

For example, the sequence of instructions, **add** R2, R1, R1 ; **add** R1, R3, R3, have write-after-read hazard. The parallel composition of two corresponding ACSR processes is as follows:
$\{ \text{R2}, \text{R1}_1, \text{R1}_2\} : \text{Done} \parallel \{ \text{R1}, \text{R3}_1, \text{R3}_2\} : \text{Done}$
Since R1 is an abbreviation of the collection for $\{\text{R1}_1, \text{R1}_2, \text{R1}_3, \text{R1}_4, \text{R1}_5, \text{R1}_6 \}$, the resources R1_1 and R1_2 appear in both the left and right processes of parallel composition operator. Thus, the above parallel process has resource conflict, that is, the process is equivalent to NIL by expansion law.

Execution of a ToyP Program. Given a ToyP program, the ToyP processor executes as many instructions as possible at each instruction cycle. To model such execution behavior, we define an indexed set of ACSR processes, called Super-Insts(PC) where PC is an index variable denoting the memory location of the current instruction. As shown in Figure 1, Super-Insts(PC) specifies the possibilities of executing three instructions, two instructions or one instruction. The choice among these three is explained in the next paragraph. After one time unit, Super-Insts(PC) determines whether or not the next instruction can be executed as specified by next Insts(PC+12). If there exists data conflict between a not-yet completed instruction and the next instruction, the execution of the next instruction is delayed. This delay possibility is specified by the left choice in the definition of Insts(PC). If there is no conflict, the next instruction is chosen for execution.

The process Program defines the behavior of the corresponding ToyP program. Note that in the definition of Program, the process Super-Insts(PC) is defined with the Close operator with the resource set R.

The process Super-Insts(PC) has three choices. These choices represent the scheduling of ToyP's execution: issuing and executing of up to three instructions simultaneously. When the sequence of instructions contains data hazard, the corresponding ACSR term becomes NIL by Lemma 4.1. Thus, data hazard terms are eliminated during the expansion of Super-Insts(PC).

The next question is how to ensure that as many instructions as possible are executed at each cycle. This is the reason why the process Program is defined as closed Super-Insts(PC). Even after the impossible execution choice due to data hazard is eliminated, the process Super-Inst(PC) can still have multiple choices. In such case, the notion of preemption in ACSR [6] allows the selection of a choice with the most number of instructions.

One of the most useful laws for process algebras is the expansion law, which can be used to eliminate parallel operators. Given a ToyP program specification, we can use the expansion law and other ACSR laws to convert a ACSR process which does not contain any parallel operators. The resulting process describes all possible behaviors of the original ToyP program and also facilitates the analysis of temporal properties.

Example 4.1 This example illustrates how to simulate the ToyP program described in Example 3.1. By law, we have Mem(PC) || Mem(PC+4) || Mem(PC+8) = NIL, since there exists resource conflict between first and third instructions (as well as second and third instructions). Thus, the Super-Insts(PC) process has the following expression after some rewriting

Program
= [Super-Insts(PC)]$_R$
= NIL + {R1, R2, $R3_1$}:{R2}:Done next Insts(PC+8)
+ {R1}:Done next Insts(PC+4)]$_R$

Since we have the following priority relation: [R1]$_R \prec$ [R1, R2, $R3_1$]$_R$, by the law of prioritized choice, we have

Program
= {R1, R2, $R3_1$}:{R2}:Done next Inst(PC+8)
= {R1, R2, $R3_1$}:({R2}:Done || {R1, $R3_2$, $R3_3$}:Done)
= {R1, R2, $R3_1$}:{R2, R1, $R3_2$, $R3_3$}:Done

Therefore, the ToyP processor issues and executes the first two instructions simultaneously. It leaves the third instruction for the next cycle, because of data hazard.

5 Conclusion

In this paper we have presented a technique for specifying the temporal properties and resource constraints at instruction-level parallelism using ACSR. We illustrate our approach using a simple superscalar processor, called ToyP. Our approach is to consider the ToyP processor as a set of resources, such as integer registers. The resource constraints of an instruction specify a sequence of sets of resources required by the instruction, one set for each instruction cycle. Each ToyP instruction is translated to a corresponding ACSR process. A ToyP program is translated to an indexed set of ACSR processes. We obtain the timing properties of the original ToyP program by simplifying the corresponding ACSR processes using ACSR laws.

We are currently experimenting with ToyP instruction specifications using ACSR tool-kit, VERSA [2]. We also have modeled various floating-point instructions and out-of-order instruction sequence execution. We are currently investigating more complex superscalar architecture such as caches and pipelines. We also are working on the formal specifications of various branch instructions, conditional instructions as well as on automatic derivation of instruction scheduling parameters.

References

[1] J. Camilleri and G. Winskel. CCS with Priority Choice. In *Proc. of IEEE Symposium on Logic in Computer Science*, 1991.

[2] D. Clarke, I. Lee, and H. Xie. VERSA: A Tool for the Specification and Analysis of Resource-Bound Real-Time Systems. Technical Report MS-CIS-93-77, Dept. of CIS, Univ. of Pennsylvania, Sept 1993.

[3] T. Cook, P. Franzon, E. Harcourt, and T. Miller. System-Level Specification of Instruction Sets. In *Proc. of the International Conference on Computer Design*, 1993.

[4] E. Harcourt, J. Mauney, and T. Cook. Specification of Instruction-Level Parallelism. In *Proc. of the North American Process Algebra Workshop*, 1993.

[5] M. Johnson. *Superscalar Microprocessor Design.* Prentice-Hall, 1991.

[6] I. Lee, P. Brémond-Grégoire, and R. Gerber. A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems. Technical Report MS-CIS-93-08, Univ. of Pennsylvania, January 1993. To appear in *IEEE Proceedings*, Jan 1994.

[7] D. Patterson and J. Hennesy. *Computer Architecture: A Quantitative Approach.* Morgan Kauffman, 1990.

[8] R. Rau and J. Fisher. Instruction-Level Parallel Processing: History, Overview, and Perspective. *Journal of Supercomputing*, 7, 1993.

Session V: Scheduling II

Chair: Hide Tokuda
CMU

Task Scheduling for Real-Time Multi-Processor Simulations

Gaetano Borriello

Dep't of Computer Science & Engineering
University of Washington
Seattle, WA 98195

Daniel M. Miles

Flight Systems Laboratory
Boeing Commercial Aircraft
Seattle, WA 98124

Abstract

Scheduling of tasks onto multi-processors is an increasingly important problem in the simulation of avionics systems. The problem is difficult due to the many hard real-time constraints imposed on the schedule in the form of processor frame-time limits and latency requirements. In this paper, we present a solution to this real-time scheduling problem using simulated annealing techniques. The running time of the algorithm is fast enough for it to be applied in the rapid reconfiguration of simulation test benches in use at Boeing Flight Systems Laboratory. Its efficacy is demonstrated using an example with 60 tasks communicating through 1800 common blocks and scheduled onto 6 processors under 4 latency constraints which achieved a utilization factor over 95%. Such an example can be scheduled in approximately 35 minutes of CPU time on an HP-Apollo 425 workstation.

1: Introduction

Simulation is a rapidly growing part of the process of building new aircraft. The complexity of modern aircraft systems, especially their avionics, requires lengthy and comprehensive testing before the first flight occurs. In testing aircraft systems, individually and together, simulation is a very effective alternative to traditional on-ground and flight testing, in terms of both cost and safety. Unfortunately, real-time simulation of avionics systems in a multi-processor environment is a complex and difficult task. One of the many challenges is the allocation of tasks to processors in a manner that efficiently utilizes the computing capacity and also meets the timing requirements of the real-time environment. The particular environment we are concerned with statically assigns tasks to processors in a fixed execution order.

An avionics system can be thought of as a collection of interconnected processing units, called Line Replaceable Units (LRU). An LRU provides a particular function, such as an auto-pilot, that can be swapped into and out of an aircraft for servicing or maintenance. Simulations are often organized in terms of LRU tasks allowing easy insertion and deletion of tasks. As hardware is constructed, it is tested in its operating environment by insertion into the simulation replacing the corresponding software LRU. There may even be multiple software models of an LRU. An engineer checking out auto-pilot control laws may require a high-fidelity auto-pilot model with its necessarily higher complexity whereas an engineer checking engine performance needs only a simple auto-pilot model to allow the airplane to fly an acceptable flight path. In any case, hardware or software LRUs must have identical interfaces to ease the job of reconfiguring the simulation.

FORTRAN is used to implement the software LRU tasks. Common blocks are used to simulate the LRU interfaces and pass data from one task to another. To facilitate the restructuring and reordering of software, common blocks are carefully partitioned by LRU output. This is needed for modularity; if outputs of two tasks share a common block and the tasks are scheduled on different processors, then one task's data may overwrite the data produced by the other when the common block is copied to shared memory. Therefore, common blocks have a single writer and multiple readers. Common blocks reside in the local memory of the processor running the LRU task. If data in a common block is needed by a task on another processor, the common block is written to shared memory and read by the task needing the data. The cost to produce or consume a common block is determined by the size of the block, plus some additional overhead. A real-time executive on each processor coordinates the tasks by consuming data, executing the tasks, and producing data.

In addition to running the real-time executive, the host computer runs LRU tasks, the airplane simulation, and records data in real-time. This is a huge load on resources, and improvements in computing capacity are always quickly used up by expanding simulation requirements. The use of multi-processor computers has provided a substantial improvement in computing capacity, but at the same time has greatly increased the complexity of effectively managing the computing environment.

Scheduling of tasks onto processors is complicated by the presence of hard real-time constraints in the form of processor frame-times and latency requirements. A processor frame-time constraint states the period with which a processor will cycle through its assigned tasks.

0-8186-5710-3/94 $3.00

This will limit the number of tasks that can be assigned to the same processor. Latency requirements apply to data paths through a series of alternating tasks and common blocks and state that a computation (represented by the path) must be completed within a specified amount of time. Latency calculations must not only take into account the execution time of the tasks but also the costs of copying common blocks to and from memory when necessary and the relative execution order of the tasks involved.

2: Simulated Annealing

The problem has a large solution space (P^T possible solutions, where P is the number of processors and T is the number of tasks) [8]. An optimal solution is one for which no frame-time or latency are violated. Existing scheduling approaches have some deficiencies when applied to this situation [1, 2, 6, 7]. Specifically, there are three points to consider. First, a pre-runtime or static scheduling approach is needed to ensure that all constraints will be satisfied all of the time. Second, processor utilization near 100% must be achievable. Third, latency constraints must be supported (note that these stretch across a sequence of tasks and cannot be expressed as simple task deadline constraints). These three characteristics lead us to investigate heuristic approaches to solving this scheduling problem.

One possible heuristic approach would attempt to mimic the decisions of the software engineer faced with the same problem. The typical approach to allocate tasks in a multi-processor environment starts with a uni-process simulation. The engineer partitions the simulation into functional units, relying on knowledge about the data flow of the LRU tasks and the frame-times necessary to ensure the individual tasks will support the time dependent computations, such as physical control laws. When LRU tasks pass data to each other sequentially, the best solution is to order them sequentially in a processor to minimize data latency. Problems arise when data flow is not clearly sequential, or when a single processor doesn't have enough processing power to execute such a set of sequential tasks. The engineer is faced with making a best guess and then analyzing the allocation to see if it meets the requirements. Processor and LRU task frame-times are emphasized over data latency because the frame-time is often a less flexible requirement for LRU tasks, and also because it is much easier for the engineer to measure and analyze than data latency. All the engineer needs to estimate execution time are the individual execution times of the tasks. To estimate data latency, processor frame-times and data flow information are needed. When an allocation is made to multiple processors, the data flow must be traced from task to task and processor to processor in order to sum the total latency as outlined in the previous section. Minor changes in allocation can have large effects on these calculations thus making the problem difficult to solve manually.

Algorithmic approaches have the same problems. It is fairly easy to meet the frame-time requirements, but difficult to meet the data latency requirements. To solve the frame-time requirements, a bin-packing algorithm can be used. A near optimal solution for frame-times is easily obtained. However, with a fairly complex data flow, an algorithm to minimize data flow latency is not readily apparent. This is further complicated when processors have differing frame-times and are asynchronous. It is difficult to capture the decisions made by the engineer to minimize data latency and would certainly make a challenging AI project. Fortunately, there are simpler alternatives.

Simulated annealing is an algorithm that has been successfully applied to another difficult allocation problem, the optimization of VLSI component placement [3, 4, 5]. The algorithm is meant to mimic the annealing process of forming a crystal. A solution is heated to a point where molecules are moving randomly and not bonding together. Then the solution is slowly cooled, and the molecules begin sticking. If the location is a strong bond, a molecule is unlikely to move again. If the location is a weak bond, a molecule will probably come loose to try and find a better location. As the solution cools, a crystal structure is formed by molecules sticking to the locations with strong bonds.

The algorithm is very simple and relatively easy to implement. The basic idea is to generate random moves (in this case, moving a task to a different processor or another position in the execution order on the same processor) and then evaluate the allocation. The evaluation is based on the frame-time and data latency of the processors. If the new allocation is better, the move is always accepted. If the move is not better, the move is accept or rejected based on a probabilistic function of two terms, the current temperature, and the difference between the new allocation's evaluation and the old.

The temperature is determined by a cooling schedule. The cooler the temperature, the less likely it is that a bad move will be accepted. The cooling schedule begins with a high temperature, where almost all bad moves are accepted and cools to a temperature where almost no bad moves are accepted.

The key to this algorithm is in the probabilistic acceptance of bad moves. A problem's solution space typically has many hills and valleys when evaluating all the possible configurations. If only good moves are accepted, that is, those that improve the evaluation, then movement within the solution space will always be downhill. Starting from a random point in the solution space, the best possible solution would be the lowest point reachable without going uphill. The problem is the best solution might be just over the next hill. Allowing bad moves gives the algorithm the opportunity to climb the hill and find the better solution on the other side. Algorithms

with this property have been termed probabilistic hill-climbing algorithms.

3: Results

The simulated annealing algorithm has been implemented on an HP-Apollo workstation using the C programming language. In this section, we describe the results obtained on a realistic example. The annealing program outputs representations of the allocations at every step in the cooling schedule, including the beginning and end. An example allocation is given in Table I.

The rows represent the order of tasks assigned to a processor, followed by two numbers separated by a colon. The number on the left is the specified frame-time of the processor, which is equivalent to the smallest frame-time of the tasks allocated to the processor. The number to the right is the actual execution time of the tasks allocated to the processor, along with the time needed to read and write common blocks to shared memory. If the actual execution time of a processor is larger than the processor's frame-time, a frame-time overflow occurs. The amount of the overflow is the execution time minus the frame-time.

The evaluation of the allocation is given by the three numbers below the allocation table. Frame-time and latency overruns are computed similarly by first finding all violated constraints. The frame-time overrun is then the sum of all the individual processor frame-time overruns for those processors with frame-time violations. The latency overrun is generated by summing the amount by which latency constraints are violated for those constraints that are, in fact, unsatisfied. The "eval" value is computed as follows: (latency overrun * weight_factor) + frame-time overrun. The weight factor is an input parameter.

The best possible value for the evaluation is zero. As long as all frame-time and latency constraints are met, all solutions are equal as far as the evaluation is concerned. The weight_factor weights the latency constraints versus the frame-time constraints. The latency overruns typically produce much larger numbers than the frame-time overruns due to the way the overruns are computed, so the latency overruns are scaled down by setting weight_factor to values between zero and one. Also, in the real-time environment of aircraft simulation, frame-time constraints are more critical than latency constraints, and the multiplier helps push overruns to the latency evaluation.

The annealing program was run on an example having 60 tasks, 1800 common blocks, and 4 latency constraints, targeting a system with 6 processors. The annealing algorithm was executed 92000 times over 5 temperatures, taking 35 minutes on an HP 425. Sample output is given in Table II.

Processor	Tasks allocated to the processor (in order)	Frame-time: exec. time
proc0	t37 t53 t50 t39 t28 t51 t5	20:15.124 *
proc1	t29 t6 t42 t57 t19 t27 t22 t26 t56 t1	20:15.1875 *
proc2	t18 t41 t33 t36 t34 t20 t12 t30 t4	20:24.1515 *
proc3	t31 t32 t44 t13 t11 t55 t38 t14 t0	20:22.171 *
proc4	t10 t40 t54 t52 t8 t48 t23 t17 t45 t24 t49 t58 t9 t3	20:42.2305 *
proc5	t43 t46 t25 t21 t35 t7 t15 t16 t47 t59 t2	20:16.2 *

latency overrun = 620 frame-time overrun = 28.553 eval = 90.553

Table I. Sample Task Allocation

Processor	Tasks allocated to the processor (in order)	Frame-time: exec. time
proc0	t10 t55 t11 t30 t4 t12 t31 t3 t59 t5	20:19.98
proc1	t0 t51 t1 t34 t16 t35 t49 t13 t36 t33	20:19.07
proc2	t22 t27 t25 t6 t7 t38 t20 t19 t26 t21 t23	20:19.575
proc3	t8 t48 t24 t18 t14 t58 t15 t46 t28	40:39.565
proc4	t40 t17 t54 t44 t56 t9 t57 t45 t2 t32	20:19.7
proc5	t52 t53 t29 t41 t43 t37 t39 t50 t42 t47	20:19.205

latency overrun = 0 frame-time overrun = 0 eval = 0

(seed = 1, weight factor = 0.1)

Table II. Sample Output from Annealing Program

The cooling schedule for this example was arrived at by experimentation. The schedule has five temperatures, 100, 10, 1.0, 0.1, and 0.01. Using an exponentially decreasing schedule is typical of annealing implementations. Initially, a much wider range of temperatures was used, from 1000 to 0.001. By studying the evaluations at the extremes of the temperature schedule, the range was narrowed with no effect on the quality of the final solution. The upper range was lowered to 100 because at temperatures of 10 and above, task allocations were seemingly random. The 100 was left in the schedule to ensure that the algorithm covered a large portion of the solution space. In this example, the algorithm converged on a solution at temperature 0.1. This is not always the case. In about 20% of the trials, there was a small frame-time overrun at temperature 0.1 which was erased by the iterations at temperature 0.01. Since an evaluation of zero was produced with the temperature of 0.01, there is no need to use a lower temperature. In no case did an evaluation ever increase once it went to zero at the 0.01 temperature. The number of iterations at each step in the cooling schedule were similarly derived by experimentation. The cooling schedule used in the above example appears to be adequate on all our experiments to date. However, given the small running time of the algorithm, it is conceivable that the number can be tuned for each problem if many new simulation configurations (using different LRUs) are to be implemented.

The process of tuning the schedule and weight factor is problem dependent, but a schedule and weight factor that works for a given problem will probably work well for similar problems. The most important factors are the size of the problem in terms of tasks and common blocks, the number and complexity of constraints, and the number of processors available. Also, the example used for Table II is tightly constrained in terms of frame-time. No processor has more than 5% idle time. It seems obvious that if a given schedule finds a solution with a tightly constrained problem, it will also work on problems less tightly constrained. Conversely, the more constraints on a problem, the more iterations will be required to solve it, if it can be solved at all.

Since the algorithm is dependent on the number of iterations, the performance of the algorithm is critical. After some coding improvements, the runtime of a smaller example was reduced from 10 minutes to a little over 3 minutes. The runtime of the larger example above was dominated by the computations related to the large number of common blocks and scaled linearly with the number of blocks and constraints in the input. This was verified by adding large numbers of artificial latency constraints. A profiling utility was used to identify the most heavily used routines and modifications to four routines (all in the annealing evaluation routing) resulted in the performance improvement. The improvements in efficiency were almost entirely derived from substituting C language pointer addressing and operations for indexed arrays.

4: Conclusion

The problem of allocating tasks in a real-time multiprocessing environment to simulate modern aircraft is intractable, but has a suitable heuristic solution using the simulated annealing algorithm. The problem has been formalized with a set of equations defining processor frame-time and data latency as functions of task allocation. The equations are directly used to define the evaluation portion of the simulated annealing algorithm. The algorithm was run on a large example based on existing real-time aircraft simulations and shown to be approximately linear in the number of common blocks and latency constraints. Several input parameters have to be fine-tuned experimentally to yield good results, but the execution speed of the algorithm (approximately 35 minutes as opposed to several days or weeks for hand methods) is fast enough to support repeated trials and be effectively applied in the Boeing Flight Systems Laboratory.

References

[1] Baker, T.P. and Shaw, A. "The Cyclic Executive Model and ADA", Proceedings of the IEEE Real-Time Systems Symposium, Dec 1988, pp 120-129.

[2] Chetto, H., Silly, M., and Bouchentouf, T. "Dynamic Scheduling of Real-Time Tasks under Precedence Constaints", The Journal of Real-Time Systems, Vol. 2, 1990, pp 181-194.

[3] Kirkpatrick, S., Gelatt, C.D., and Vecchi, M.P. "Optimization by Simulated Annealing", Science, May 1983, pp 671-679.

[4] Mitra, D., Romeo, F., and Sangiovanni-Vincentelli, A. "Convergence and Finite-Time Behavior of Simulated Annealing", Proceedings of the 24th Conference on Decisions and Control, 1985, pp 761-767.

[5] Shahookar, K. and Mazumder, P. "VLSI Cell Placement Techniques", ACM Computing Surveys, June 1991, pp 143-219.

[6] Sprunt, B.J., Sha, L., and Lehoczky, J.P. "Aperiodic Task Scheduling for Hard-Real-Time Systems", Real-Time Systems, June 1989, pp 27-60.

[7] Versoosel, J.P.C., Luit, E.J., and Hammer, D.K. "A Static Scheduling Algorithm for Distributed Hard Real-Time Systems", The Journal of Real-Time Systems, Vol. 3, 1991, pp 223-246.

[8] Zondag, E.G. "A Static Load Distribution Strategy for Processor Pools", Memorandum INF-88-43, University of Twente, The Netherlands, 1988.

Successful Use of Rate Monotonic Theory on a Formidable Real Time System

Larry Doyle and Jon Elzey

(ITT Aerospace/Communications Division)

Abstract

The navigation payload software for the next block of Global Positioning System satellites recently completed testing. The computer program for the onboard computer is sufficiently complex to expose almost every issue that has been put forward in rate monotonic theory. The success of this effort demonstrates the utility of the theory in this type of application. The system designed required the processor to perform a highly diverse set of hard deadline real-time functions. This design would have been difficult or impossible prior to the development of rate monotonic theory. The use of utilization bounds has important advantages from a software engineering point of view. The problems of insuring schedulability over the course of development and verifying the schedulability of the finished system are discussed.

Background

Rate monotonic scheduling theory has been successfully applied to the development and testing of complex real time software for an embedded space vehicle application. The project development occurred over the same period of time that much of the material on rate monotonic scheduling was being published. Thus, it was an opportunity to apply these ideas very soon after publication to a large project with cost and schedule commitments. Because the system design required the processor to perform such a complex mix of functions, it depended heavily on the application of this theory.

The software was developed for the Navigation Payload in the NAVSTAR Global Positioning System (GPS) Replenishment Satellites program. These satellites will be launched during the late 1990's as the current generation of GPS satellites is retired. The new satellites have significantly greater functionality than their predecessors.

Work began on this project in the spring of 1987 and completed final qualification testing in the fall of 1993. Hardware production and related work will extend into the 1990's. The initial phase of development was a competitive contract that led to the award of the current development contract. In the initial phase, a brassboard prototype was developed which supported a core subset of functions for the Navigation Payload. The prototype provided a proof of concept and some execution time benchmarks without which the risk of the ensuing development would have been unacceptable. Full scale development was completed on the next phase. Although some time, perhaps a year, can be attributed to the transition between two contracts, the extended development time reflects the scope and complexity of the effort.

The Global Positioning System

The Global Positioning System provides navigation and time signals to suitably equipped users on a global basis for position, velocity and time determination. It also provides a nuclear event detection capability.

GPS is composed of a User Segment, a Control Segment and a Space Segment. The User Segment consists of user navigation receivers that receive and process the satellite downlink. The Control Segment consists of ground stations responsible for monitoring the space vehicles, supplying the space vehicles with updated Navigation Data, and ensuring proper space vehicle operation. The Space Segment is a constellation of twenty-four space vehicles in six orbital planes, providing twenty-four hour coverage worldwide. On the space vehicle, the Navigation Payload performs the Space Segment role in the navigation mission and provides communications for the nuclear detection mission.

Because these satellites are used for aircraft and other navigation, the integrity and reliability of the system are critical. If the satellite were to broadcast erroneous signals or data, the consequences could be disastrous. In this type of hard deadline, real-time system, failure of a task to complete in time could have unpredictable results. The

0-8186-5710-3/94 $3.00

hardware often enters an undefined or undesirable state if the software fails to meet a deadline. It was therefore critical to insure the reliability of the real time design.

The Navigation Payload Mission Processor

Within the Navigation Payload, the Mission Data Unit (MDU) serves as a central interface point to numerous other space vehicle subsystems. These include the Telemetry, Tracking and Control System, the Nuclear Detection System, the Spacecraft Processing Unit, an inter-satellite crosslink for communications and ranging, and an L-band downlink. The MDU also contains specialized hardware for timing control, modulation control, navigation, and communications security. The MDU includes the flight software which is the subject of this paper. This is an Ada program, supported by the EDS Scicon XD Ada cross-compiler and linker. The embedded MDU processor, called the Mission Processor, is a Marconi MAS281 which is an implementation of the MIL-STD-1750A architecture with memory mapping.

The computer program in the Navigation payload is large and complex compared to other spacecraft software. It performs a wide variety of loosely coupled functions. There are numerous communication functions which involve data buffering, bit-packing, forward error correction coding, error detection and cryptography. In addition, the software performs a phase locked loop that maintains the highly accurate timing signal to the navigation users. The requirement to function autonomously for 180 days dictates that the spacecraft will perform many computationally intensive functions which are currently performed by the Control Segment. These include monitoring the integrity of the navigation and timing information, estimating the satellite's orbital parameters from inter-satellite range measurements, and maintaining the synchronization of the GPS constellation using the inter-satellite crosslink.

The software deadlines result from the many real-time interfaces that must be serviced continously. The I/O architecture includes twenty-seven interfaces which use nineteen interrupts with rates up to one kilohertz. All six spare 1750A interrupts are used, three of which are multiplexed. Thirty-two I/O ports are allocated to the register I/O address space, and twenty I/O ports to a memory mapped I/O page in one of the address states.

The processor memory architecute requires the program to be partitioned into address states which have separate logical adress spaces. The Navigation Payload software maps five 1750A logical address states to 448K 16-bit words of physical memory. Additional memory is allocated to store data received from the Control Segment. The XD Ada runtime system requires a separate Ada main program in each address state. Global shared data provides inter-state communication and the XD Ada implementation of the classical semaphore provides inter-state synchronization.

Perhaps the most distinguishing feature of the software design is that there are fifty-one tasks. Some Ada designs may have a large number of tasks which serialize access to data. Although this is the case with some tasks, most of these tasks manage asynchronous activities with deadlines. Five tasks are XD Ada direct interrupt handlers. Two additional interrupt handlers are the only software coded in assembly language. Other tasks are allocated as application processing threads or as servers. Tasks have deadlines ranging from a few milliseconds to several minutes. Because the Mission Processor is required to resume operation after a processor reset, some checkpointed data are not altered by Ada elaboration. Initialization following Ada elaboration and task activation is distributed over the tasks. The XD Ada runtime system provides task scheduling in the Ada preemptive model without time slicing. Interrupt handlers may propagate events through rendezvous or shared data. Lower rate tasks schedule themselves by delay statements or by rendezvous with a higher priority task which hands off a large computational job with a later deadline. Servers wait for a rendezvous with a client. From the point of view of data flow and functional processing, the software contains several pipelines with delays corresponding to rate monotonic scheduling deadlines.

Since this large number of tasks incurs penalties in memory usage, runtime overhead, and the comprehensibility of the system, the design was often reviewed to see if the number of tasks could be reduced. Most of the shared data structure tasks have been optimized away. In the final design, only a small reduction would be possible at the expense of significantly poorer modularity. Insuring schedulability of such a large number of tasks was a significant problem.

Although much detail has changed since the prototype, the basic software architecture and use of Ada tasking have not. One source of change was the evolution of many functional and interface requirements, another was a change of compiler vendors. Development began with a compiler that used the Ada rendezvous as the means of inter-address state communication and synchronization. In early 1991, the program transitioned to the XD Ada compiler. The difference in interrupt models induced changes in interrupt handlers. Interrupts are now handled as described in reference [3]. The alternate methods for dealing with 1750A memory

mapping caused substantial task re-allocation, including proliferation of agent tasks to transport service requests across address states between clients and servers.

Software development followed substantially the methodologies described in reference [10]. In addition, unbounded priority inversion was prevented by design practices similar to those described in [11], [4], [5] and [6].

Insuring Schedulability throughout Development

Reference [4] gives two groups of techniques for guaranteeing schedulability. The first group of techniques, derived from references [7] and [11], is based on computing utilization bounds. The second group of techniques verifies schedulability by determining the response time of each task. In these techniques, there is a tradeoff between complexity and pessimism. The simplest techniques tend to yield overly pessimistic results. As the techniques become more complex, they yield increasingly realistic results. All of the techniques for determining response times are more difficult and require more data than the techniques for computing utilization bounds.

Application of the most accurate techniques to a system of this complexity would be far too costly and would have delayed the schedule. In addition, since execution time budgets must be continuously monitored, it is important that the data required to do this monitoring not be too complicated. The most economical analysis will do just enough to prove schedulability and no more.
Rate monotonic analysis requires a global view of the entire processor. However, good design dictates that the program be decomposed such that each piece can be designed with limited knowledge of the other pieces. On a complex system, it is important to be able to deal with the global properties of the system at a higher level of abstraction. Utilization bounds are in the spirit of information hiding in that only a small amount of information about each task need be exposed. Information hiding has been shown to be an important element of software productivity [1]. This "separation of concerns" is also discussed in [11]. As a result of these considerations, only utilization bound techniques were employed.

Execution time budgets were frequently exceeded at every stage of the development cycle. This required reallocating budget or performing some optimization or both. When an optimization is required, an abstract, global view of schedulability is essential in deciding which of the potential optimizations will best address the problem. The fact that optimization was needed so frequently points out the importance of evaluating execution time estimates and measurements as they become available.

Utilization was continuously updated as the design matured and execution time estimates became more accurate. In the early phases, a single utilization bound was used to assign initial execution time budgets. Since processing times always seem to increase as a software project progresses, the fact that a single utilization is conservative is an advantage. As the design matured, the single bound technique yielded too pessimistic a result and more complex techniques were employed.

Use of a single utilization bound requires that all tasks are strictly periodic. If events have a burst rate higher than the average rate, this shortest period must be used in the analysis. If the deadline is less than one full period, you must either use a lower bound, inflate the execution time, or assume a period for the task equal to the worst case deadline. Because of these considerations, the utilization that is computed for a single bound comparison is much higher than what is actually the case.

Because the Mission Processor has many cases of both shorter-than-period deadlines and high burst rates that are significantly higher than the average rate, this system would not meet the single bound criteria. It was necessary to use multiple bounds. Reference [4] describes use of utilization bounds for each event when the deadlines are within the period. The utilization of task i, f_i, is computed as follows:

$$f_i = \sum_{j \in Hn} \frac{C_j}{T_j} + \frac{1}{T_i}\left(C_i + B_i + \sum_{k \in H1} C_k\right)$$

where C is compute time, T is the period, B is blocking, Hn is the set of tasks with a shorter period than task i, and H1 is the set of tasks that must execute in order for task i to complete. This utilization is then compared to the appropriate bound.

A major simplification in the use of this technique is that it is not necessary to perform this for every event. At the expense of being slightly more conservative, the equation above can modified to compute utilization for a set of tasks. Tasks can be divided into sets that span ranges of rates and utilization can be computed for each set. With a small number of sets, most of the simplicity of a single bound is retained. Selecting the sets carefully eliminates the large discrepancy between computed and actual utilization which occurs with a single bound. For most of the design process, just two bounds were used.

This was later extended to three. Utilization for a set of tasks, H2, can be expressed as:

$$f_{H2} = \sum_{j \in Hn} \frac{C_j}{T_j} + \frac{1}{\min(T_k)_{k \in H2}} \left(\max(B_k)_{k \in H2} + \sum_{k \in H2} C_k \right)$$

Comparing this utilization to the appropriate bound can then prove the schedulability of the set of tasks. The periods, T_k , in H2 are the burst rate periods but the periods, T_j , in Hn are the average over $\min(T_k)$. This eliminates the excessive conservatism of a single bound.

Verifying Schedulability of the Finished System

To some extent, the schedulability of a system is demonstrated by the various functional tests. However, on a complex system it is not practical to figure out how well the test cases prove that system is schedulable in every possible condition. Therefore, a final analysis was performed with measured execution times. On a finished system, there is the option of measuring the response times directly. This was considered but there were instrumentation problems with several events. Therefore, the same utilization bound technique that was used during development was used in the test. This required accurate execution time measurements.

The first attempt was to use end-to-end execution times as though they were the actual times. Test scenarios were constructed which attempted to minimize the activity which would preempt the task that was being measured. The HP Software Performance analyzer was used to measure the time between a synchronization event, such as a delay expiration or interrupt, and the completion of processing. This device builds a histogram of durations which occur between two events. From this, the longest response time in a test scenario can be measured. The test scenario is designed so that it forces at least one occurrence of the longest execution path of the task being measured. However, the nature of the system is such that preemption could not be sufficiently minimized.

The XDAda runtime system has a pointer to the task control block of the currently running task. This pointer is changed if, and only if, there is a context switch. With the HP State Analyzer, we could trigger on a desired event and measure the exact time between every context switch until the completion of the response to the event. This had the virtue that all runtime overhead was included in the measurement. It also provided the blocking time measurements. The disadvantage is that substantial knowledge of the program flow was required to unravel which task executions were part of the response. This turned out to be less difficult than it appeared and it had the unexpected benefit that it uncovered some design flaws.

Conclusion

This project was a successful application of rate monotonic theory to the design, analysis and testing of a complex real-time system. It would have been very difficult, perhaps impossible, to implement this system without preemptive scheduling. Throughout the project, concerns about the use of Ada and preemptive scheduling in hard deadline systems were published [9], [8], [2]. There were also reservations within our own organization. None of the difficulties were significant compared to the advantages of the approach.

What is most important, the methodology used makes it possible to deal with the overall schedulability of a complex real-time system at a higher level of abstraction while maintaining the loose coupling and decentralized design of the pieces.

Several major issues were only touched on in this paper. The design optimizations which were required provide important insights. The hardware/software tradeoffs that lead to this design suggest some trends in real-time systems. Some important techniques in the use of Ada were devised. Economically obtaining execution time estimates throughout the design was a challenge. These may be the subject of future papers.

References

1. Boehm, B, Improving Software Productivity, Computer, September, 1987.

2. Cooling, J.E., Software Design for Real-Time Systems, Chapman and Hall, 1991.

3. Jennings, P., Ada Interrupt Handlers for Hard Real-Time Systems, International Workshop on Real-Time Ada Issues, ACM Press, 1990.

4. Harbour, M.G., Klein, M.H., Rayla, T., Pollak, B., Obenza, R., A Practioner's Handbook for Real-Time Analysis, Software Engineering Institute, Kluwer Academic Publishers, 1993.

5. Leinbaugh, D.W., Guaranteed Response Times in a Hard-Real-Time Environment, Transactions on Software Engineering,Vol. 6, No. 1, Jan. 1980.

6. Levine, G., Control of Priority Inversion in Ada, Ada Letters, Vol. VIII, No. 6, Nov./Dec. 1988.

7. Liu and Layland, Scheduling Algoritms for Multiprogramming in a Hard Real Time Environment, J. ACM, Vol. 20, No. 1, 1973.

8. Locke, C.D. and Vogal, D.R., Problems in Ada Runtime Task Scheduling, International Workshop on Real-Time Ada Issues, ACM Press, 1987

9. McCormick, F., Scheduling Difficulties of Ada in the Hard Real-Time Environment, International Workshop on Real-Time Ada Issues, ACM Press, 1987.

10. Nielsen, K.W., and Shumate, K., Designing Large Real-Time Systems with Ada, Communications of the ACM, August, 1987.

11. Sha and Goodenough, Real-Time Scheduling Theory and Ada, Computer, April 1990.

Temporal Protection in Real-Time Operating Systems

Cliff Mercer*, Ragunathan Rajkumar+ and Jim Zelenka*
*Department of Computer Science
+Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Real-time systems manipulate data types *with inherent timing constraints.*[1] Priority-based scheduling is a popular approach to build hard real-time systems, when the timing requirements, supported run-time configurations, and task sets are known *a priori*. Future real-time systems will need to support these hard real-time constraints but in addition (a) provide friendly user and programming interfaces with audio and video data types (b) be able to communicate with global networks and systems on demand, and (c) support critical command and control services despite potential risks introduced by such added flexibility and dynamics. In this paper, we argue that temporal protection mechanisms can be as beneficial in these systems as virtual memory protection. The processor reservation mechanism that we have implemented in Real-Time Mach, for example, provides guaranteed timing behavior for critical activities.

1. Introduction

In real-time systems, the correctness of a computation depends upon both its logical and temporal correctness. As a result, earlier real-time systems were often hand-crafted in order to meet stringent timing constraints. Recently, more flexible priority-based scheduling approaches have become popular [3, 6, 13]. However, the design of such systems still requires *a priori* knowledge of tasks and their timing requirements, and therefore these systems tend to be very static in nature. Many recent trends indicate that future systems increasingly need to be much more dynamic in nature:

- Real-time applications are becoming more pervasive and complex. For example, the next generation of naval systems are expected to support many data types including analog, discrete, graphics, audio, video and voice, with integrated communications and control with low latency requirements. This trend has been accelerated by two related mainstream factors. First, the explosive surge of multimedia applications has literally brought time-critical data types (audio and video) to the desktop. Secondly, the continuing growth of computing power at ever falling prices enables more and more applications, with multitasking becoming a natural candidate to use up available cycles.
- The advent of high-performance networks such as ATM opens up new applications with high bandwidth, low latency, and guaranteed service requirements. These applications include tele-medicine, distance learning, advanced air traffic control, sophisticated defense systems and networked patient monitoring. In these applications, any overload conditions because of dynamic requests and/or connections must not disrupt their basic mission.
- There is a rush towards universal connectivity and access, where a piece of information (or a person) is just a call away. Such high-degree of connectivity between systems and networks (both via cable and wireless networks) provides global access to information databases.

[1]This research was supported in part by the U.S. NRaD under contract number N66001-87-C-0155, by the Office of Naval Research under contract number N00014-84-K-0734. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of NRaD, ONR, or the U.S. Government.

0-8186-5710-3/94 $3.00 © 1994 IEEE

Future systems would therefore be hard pressed not to exploit such information availability and reachability. In addition, such information accesses may need to be set up dynamically on user demand.

The above trends are expected to result in flexible, friendly, dynamic and informative systems but this flexibility and accessibility bring about new problems such as issues of security and privacy. In addition, real-time connections (such as multimedia) must be established online. These multimedia interfaces require real-time behavior, but not at the cost of adversely affecting those critical activities with hard real-time constraints. In this paper, we argue that robust temporal protection mechanisms are needed to supplement and complement spatial protection mechanisms in order to achieve this goal.

2. Protection Mechanisms for Real-Time Programs

Early on, real-time systems were often built without virtual memory protection because the stochastic nature, long delays and potential overhead introduced by demand paging were considered incompatible with real-time requirements. However, processors and memory have become faster, and real-time operating systems [4, 11, 16, 17] have become more sophisticated. Any associated overhead of virtual memory management is now considered to be worth the address space protection enforced across process boundaries. With address space protection, logical misbehavior on the part of one process (such as the use of incorrect address pointers) does not necessarily mean that an entire processor will fail. We refer to such protection mechanisms as *spatial protection*.

Traditional non-real-time operating systems also provide a simple notion of *temporal protection* with fairness as a primary motivation. The scheduler in time-sharing systems typically uses a multi-level feedback queueing mechanism such that a process executing for a long time typically has its priority lowered in order to let other waiting processes execute [5]. Hence, even a process which enters an infinite loop can normally be stopped or killed when its scheduled time quantum expires and another process is scheduled.

In real-time systems, where timeliness of critical activities (and not fairness) is the primary motivation, the issue of temporal protection needs to be substantially re-considered. For example, consider the use of fixed priority scheduling approaches such as rate-monotonic scheduling [7, 15] in building real-time systems. Under a fixed priority scheduler, a process which enters an infinite loop will preempt all its lower priority tasks, which can then never run again. Often, the only recourse is to reboot the machine.

Spatial virtual memory protection has been adapted to real-time systems by providing the ability to lock down memory pages. Similarly, it is reasonable to expect that temporal protection schemes need to be adapted to real-time systems in general and priority-driven real-time systems in particular. We believe that such temporal protection mechanisms are critical to support the flexible and dynamic application environments described earlier. In the next section, we consider temporal protection schemes in detail.

2.1. Temporal Protection for Real-Time Programs

Many real-time operating systems support a multitasking environment for its inherent modularity, ease of program development and debugging, and programming (as well as conceptual) compatibility with traditional operating systems such as Unix [1]. The timing behavior of a real-time process in this multitasking environment depends upon its own behavior *and* its level of resource-sharing with other processes. Resources shared by processes can be either physical or logical. Physical resources shared across processes include the CPU, buses, networks, memory pages, memory heap, I/O interfaces etc.

Logical resources can include servers, shared queues, communication buffers, etc.

Scheduling theory for processors [7], buses [14] and networks provides the means to determine whether a set of tasks using a physical resource can meet its timing requirements. Similarly, synchronization protocols [13, 12, 2] provide the ability to analyze the needs of real-time tasks to share logical resources. However, these analytical techniques must necessarily make assumptions such as the worst-case execution time of a task, the maximum duration of a critical section, or the maximum bus transaction time. If these assumptions are violated, undesirable consequences can occur. In static systems, it may be relatively easier to ensure at development time that these assumptions are indeed satisfied. However, as real-time systems and applications become more dynamic and flexible in nature, the robustness of the system's ability to deliver its critical functionality may be compromised by errors and violations in a relatively new and untested process.

3. Guaranteed Processor Reservation for Real-Time Programs

Processor and memory sharing are two critical pieces which can substantially affect (and dominate) the timing behavior of a real-time program. We have been investigating an operating system abstraction called *processor reserve* [8, 9] to provide temporal protection to a real-time process at the level of CPU sharing. In this abstraction, we view processor capacity as a quantifiable resource which can be reserved like physical memory or disk blocks. A processor reserve represents a claim on processor capacity over time (e.g. 10 *ms* of computation time out of each 50 *ms* of wall-clock time). An admission control policy determines whether a reservation request is accepted or not, and once the processor reservation is established, it is scheduled and enforced by the operating system. Together, reservation and the enforcement mechanism provide a scheduling firewall which protects reserved programs from outside interference in much the same way as memory protection isolates a program address space from access by other programs.

Our processor reservation scheme has been implemented in Real-Time Mach [17]. Real-Time Mach supports a priority-driven paradigm to schedule real-time tasks[2]. Each processor reserve is assigned a rate-monotonic priority based upon its requested rate of usage and the processor is still scheduled on the basis of fixed priorities[3]. The reservation scheme includes an admission control policy to prevent overload and a mechanism to accurately measure computation time consumed by programs. In addition to measuring computation time usage, the reservation mechanism enforces computation time limits over the short-term in order to ensure that a program which attempts to use more computation time than its allocation does not interfere with the timing behavior of other programs.

3.0.1. Experimentation with Processor Reservation

We now describe an application built on our reservation scheme. The application consists of a number of instantiations of a QuickTime video player [18], each of which displays a video stream on the screen. Each program reads a short video clip and then begins to output frames to the screen using a memory-mapped frame buffer. The video resolution is 160x120 with 8 bits of color. The program applies a noise filter to each frame before it is displayed. By itself, one instantiation of the program can run at 23.2 frames per second on a 486-based machine.

---

[2]Other CPU scheduling policies such as round-robin are also provided as dynamic configuration options.

[3]It is relatively easy to extend this scheme to use dynamic priorities based on earliest deadline scheduling.

When we run two instantiations of the program under a time-sharing policy, each program averages 11.6 frames per second. Under this policy, the programs get variable service: first one video stream gets preference from the time-sharing scheduler and then the other, and they alternate getting better and worse frame-update service as their priorities change in the multi-level feedback queue.

Our reservation system allows us to go further in controlling the timing execution behavior of these two programs. If we consider one of the programs as the "focus" in the same way a video teleconference has one video stream as the "focus," we can reserve more processor capacity for that stream at the expense of the second stream. For example, when we give one video stream a reservation of 80% of the processor (e.g. 80 ms every 100ms) and allow the other to consume the remaining processor capacity, we get 18.6 frames per second on the "focus" video stream and 4.2 frames per second on the other stream.

3.0.2. Processor Reservation Manager

Fine-grained feedback on performance and the status of the reservation can help the application adapt to its own behavior and to the behavior of other parts of the system. Also, reservations may be changed by forces external to the reserved program, and the program must be informed of the change so that it can adjust its behavior. A "reservation manager" that manages the reservations on a system based on user input via a reservation user interface might make such external reservation adjustments.

3.0.3. Processor Reservation in Distributed Systems

Processor capacity reserves can support reservation in distributed real-time systems by having each reserve contain reservations for various resources around the distributed system. Then messages containing requests for remote service will contain these "sub-reserves" which can be used to "charge" the remote service. Another aspect of reservation in distributed systems concerns the reservation of communications protocol processing on each of the hosts [10]. We are currently investigating other applications of processor reservation in user-level schedulers and dedicated bandwidth for critical activities.

4. Conclusion

Real-time systems need predictable timing behavior, and predictability is often achieved by exploiting the *a priori* knowledge of supported system functionality. These systems therefore tend to be static in nature. However, due to recent trends towards multimedia applications, high-performance networking and wide connectivity, it can be expected that future real-time systems will support a highly dynamic mix of applications and connections. These flexible and dynamic systems can be susceptible to errors and misbehavior on the part of some task(s) and/or network/bus traffic. It is highly desirable that protection mechanisms be available in these systems to ensure that critical functionality is still provided by preventing temporal interference from other activities.

Address space protection offered by virtual memory provides a logical fence between processes. We similarly argue that temporal protection mechanisms are also crucial fences that need to be built between the timing behavior of critical real-time activities. One of our abstractions for such temporal protection is called the *processor reserve*. This abstraction ensures that a real-time task is guaranteed a required fraction of the processor at a certain rate. This abstraction has been implemented in Real-Time Mach where tasks with guaranteed reservations are themselves scheduled using rate-monotonic priority assignment. We are currently investigating other temporal protection mechanisms in the management of memory, display and storage.

References

1. M. J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, 1986.

2. Baker, T. "Stack-Based Scheduling of Realtime Processes". *Journal of Real-Time Systems 3*, 1 (March 1991), 67--100.

3. Burns, A. "Scheduling Hard Real-Time Systems: A Review". *Software Engineering Journal* (May 1991), 116-128.

4. K. Jeffay, D. L. Stone and F. D. Smith. "Kernel Support for Live Digital Audio and Video". *Computer Communications (UK) 15*, 6 (July-August 1992), 388-395.

5. Leffler, S. J., McKusick, M. K., Karels, M. J. and Quarterman, J. S. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989.

6. Lehoczky, J. P. "Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines". *IEEE Real-Time Systems Symposium* (Dec. 1990).

7. Liu, C. L. and Layland J. W. "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment". *JACM 20 (1)* (1973), 46 - 61.

8. C. W. Mercer and S. Savage and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. Tech. Rept. CMU-CS-93-157, School of Computer Science, Carnegie Mellon University, May, 1993.

9. C. W. Mercer and S. Savage and H. Tokuda. Processor Capacity Reserves: An Abstraction for Managing Processor Usage. Proceedings of the Fourth Workshop on Workstation Operating Systems (WWOS-IV), Oct, 1993.

10. C. W. Mercer, J. Zelenka and R. Rajkumar. "Resource Reserves for Operating System Protocol Processing". *Submitted for publication* (January 1994).

11. *IEEE Standard P1003.4 (Real-time extensions to POSIX).* IEEE, 345 East 47th St., New York, NY 10017, 1991.

12. Rajkumar, R., Sha, L., and Lehoczky J.P. "Real-Time Synchronization Protocols for Multiprocessors". *Proceedings of the IEEE Real-Time Systems Symposium* (1988), 259-269.

13. Sha, L., Rajkumar, R. and Lehoczky, J. P. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization". *IEEE Transactions on Computers* (September 1990), 1175-1185.

14. Sha, L., Rajkumar, R. and Lehoczky, J. P. "Real-Time Scheduling Support in Futurebus+". *IEEE Real-Time Systems Symposium* (Dec. 1990).

15. Sha, L., Rajkumar, R. and Sathaye, S. "Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems". *IEEE Proceedings Journal* (January 1994).

16. Stankovic, J. A. and Ramamritham, K. The Design of the Spring Kernel. Proceedings of the Real-Time Systems Symposium, Dec, 1987.

17. H. Tokuda and T. Nakajima and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. Proceedings of USENIX Mach Workshop, Oct, 1990.

18. H. Tokuda and T. Kitayama. Dynamic QOS Control Based on Real-Time Threads. ART Project Technical Memo, School of Computer Science, Carnegie Mellon University.

Session VI: Operating Systems II

Chair: Keith Marzullo
UCSD

On Latency Management in Time-Shared Operating Systems*

Kevin Jeffay

University of North Carolina at Chapel Hill
Department of Computer Science
Chapel Hill, NC 27599-3175
jeffay@cs.unc.edu

Abstract: The design of general purpose operating systems impose constraints on the way one can structure real-time applications. This paper addresses the problem of minimizing the end-to-end latency of applications that are structured as a set of cooperating (real-time) tasks. When applications are structured as a set of cooperating tasks the time required for data to progress from an input task to an output task is a function of the number of the tasks that handle the data and the deadlines of individual tasks. We present an integrated inter-process communication and scheduling scheme that can be used to minimize the end-to-end latency of multi-threaded applications. Our approach is to provide the scheduler with information on the inter-process communication interconnections between tasks and to use this information to guarantee an end-to-latency to applications that is simply a function of the timing properties of the application and not its task structure. This scheme has been implemented within the YARTOS kernel and is presently being ported to the Real-Time Mach kernel.

1. Introduction

Multimedia applications that process streams of live and stored audio and video are stimulating research on the integration of real-time computation and communication services into general purpose, time-shared operating systems. While much is known about the scheduling and resource allocation problems that comprise the formal underpinnings of such services, techniques for implementing and using existing algorithms, in particular within the context of general purpose operating systems, have received relatively little attention.

In this note, we describe a problem that arose during the implementation of an experimental desktop video-conferencing system [4, 5]. Abstractly, the problem is that of minimizing end-to-end latency in real-time applications that consist of a set of cooperating tasks or threads. Here latency is defined as the difference between the times at which input data is first made available to an application thread and the time at which an application thread performs an output operation based on the input data. The thesis of this work is that by providing the kernel with information on the task structure of real-time applications, one can both dramatically reduce the worst case end-to-end application latency and employ relatively simple scheduling algorithms to provide real-time response to individual tasks.

The following section motivates the end-to-end latency problem using an idealized version of our video-conferencing system as an example. Section 3 outlines a real-time message passing service that we constructed within the YARTOS (Yet Another Real-Time Operating System) kernel [7]. We show how this service reduces worst case end-to-end latency and how it can be efficiently implemented. The YARTOS message passing service is currently being ported to the Real-Time Mach kernel [11] and will form the basis for a comparative study of the real-time performance of the YARTOS and RT-Mach thread models.

2. The End-to-End Latency Problem

Real-time computations require bounded response times. In general, by employing results from the real-time scheduling literature (*e.g.*, [10]), for relatively simple

* Supported in part by grants from the IBM Corporation, the Intel Corporation, and the National Science Foundation (numbers CCR-9110938 and ICI-9015443).

0-8186-5710-3/94 $3.00 © 1994 IEEE

models of computation, it is possible to (1) determine conditions under which it is theoretically possible to guarantee that an invocation of a task will complete execution by a certain point in time, and (2) allocate resources within an operating system to ensure that an invocation of a task actually achieves its response time bound.

Often, it is desirable to guarantee a response time to a collection of cooperating tasks that execute in concert to realize some application. For example, consider the video processing portion of a desktop videoconferencing application. The goal of this application is to acquire, compress, and transmit a logically infinite sequence of digitized video frames across a network. The application is composed of the following (idealized) tasks:

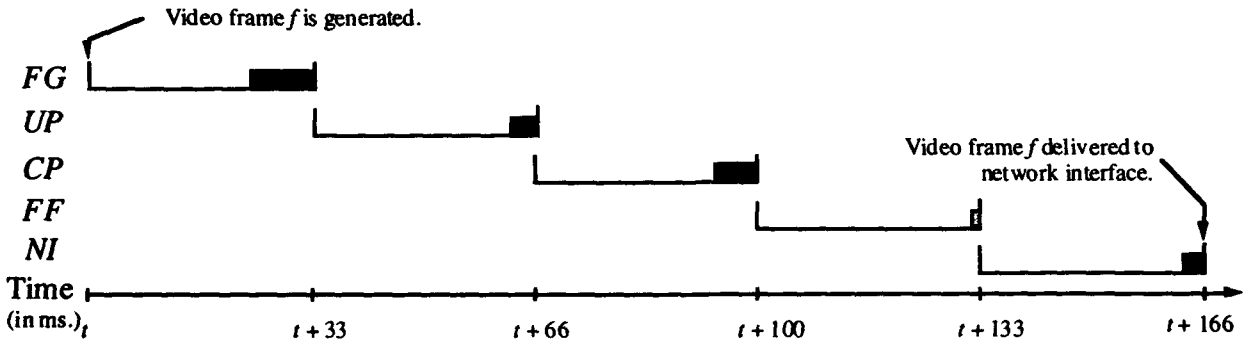
- *FG* — a task to control a frame-grabber that digitizes video frames generated by a camera,
- *UP* — a task to invoke user programs on the digitized frames for any user-level image processing that is desired (*e.g.*, for feature extraction and notification),
- *CP* — a task to compress the digitized frame,
- *FF* — a task to format and fragment the compressed frame(s) into network packets for delivery across a network, and
- *NI* — a task to control the network interface hardware.

These tasks cooperate to form a simple pipeline. Every video frame generated by the camera is digitized, processed by the user, compressed, formatted, and delivered to the network interface for transmission across the network.

In order for this conferencing application to be effective, two real-time constraints must be met. First, every video frame that is generated by the camera must make it through all stages of the pipeline and be delivered to the network interface. Second, the end-to-end latency of each frame — defined as the difference between the time the frame arrives at the network interface and the time the frame was generated — must be kept to a minimum. Since current video cameras and frame-grabbers generate data at regular, periodic intervals, the first constraint is easily satisfied by implementing each stage of the pipeline as a periodic task and using any number of real-time scheduling algorithms from the literature to schedule the tasks. The second constraint is not so easily satisfied.

An (NTSC) video frame is generated, and enters the pipeline, every 33.3 ms. In our implementation of the above video pipeline, every task has a period of 33.3 ms. Since there are 5 stages in the pipeline, the worst case end-to-latency of a video frame is 166.6 ms. The worst case occurs when each invocation of each task completes as late as possible within its period and stage $i + 1$ of the pipeline is not invoked until stage i has completed as shown in Figure 1. Whether or not the worst case actually occurs will depend on factors that are beyond the application writer's control such as the magnitude of the total system workload (*e.g.*, the number of other real-time and non-real-time tasks sharing the processor).

Note that the latency bound of 166.6 ms is really an artifact of the pipeline implementation of the video application and is not fundamental to the conferencing



Worst case end-to-end latency for a single frame of video.

Figure 1

problem itself. For example, consider an implementation of the conferencing application that combines functions *FG*, *UP*, *CP*, *FF*, and *NI* into a single task. If the task had a period of 33.3 ms, then all video frames that are generated will be delivered to the network interface (assuming the system is still schedulable) and the worst case end-to-end latency of each frame will be no more than 33.3 ms.

The problem is that it is not always possible (or desirable) to collapse all application and system functions into a single task. In particular, when executed on top of a general purpose operating system, many of the real-time application's functions (such as input and output), are implemented by operating system system calls or servers (and associated device drivers) and are shared with other applications.

The challenge therefore is to support the pipeline model of application design and execution while not incurring the penalty inherent in the straightforward realization of the pipeline. Specifically, we would like to structure the conferencing application as a series of cooperating tasks and maintain a worst case end-to-end latency bound of 33.3 ms — the period of a single stage of the pipeline.

Note that in principle this should be possible since the total amount of computation (ignoring operating system overhead) performed by the single and multi-task implementations of the application are the same. The only difference is that in the multi-task implementation of the application, several video frames may be processed simultaneously (*i.e.*, several video frames may be in the pipeline at any one time).

3. A Real-Time Message Passing Service

Our solution to the problem of minimizing end-to-end latency is to make the pipeline structure of the conferencing application known to the kernel and to use this information to schedule the stages of the pipeline as if they were part of a larger sequential program. This technique has been implemented as part of the message passing system in the YARTOS kernel [7]. We begin with an overview of the YARTOS programming model.

The YARTOS kernel supports a simple data-flow model of real-time computation. Briefly, applications are composed of tasks, resources, and ports. Tasks are threads of control, resources are shared abstract data types, and ports are queues for messages. Tasks communicate with other tasks by sending messages to ports. Each port is bound to a unique task. When a message is sent to a port, the kernel schedules the task bound to the port so that the message will be consumed before a deadline defined by the rate at which the message sender emits messages. The deadline is chosen so as to ensure that all messages from this sender can be processed in real-time (*i.e.*, without any buffering). (The YARTOS programming model is explained in greater detail in [9]. The scheduling algorithm used in the kernel is described in [6].)

When tasks and ports are created, the kernel constructs a directed graph of all possible communication paths. When a message is sent from task T_i to task T_j, the deadline for task T_j is computed using the time of T_i's most recent invocation as invocation time for T_j. That is, tasks T_i and T_j are scheduled as if they were invoked simultaneously — as if they were a single task.

For example, assume task T_i is invoked at time t and has a deadline at time $t + p$. T_i executes sometime during the interval $[t, t+p]$ and sends a message to task T_j. No matter when the message is actually sent to T_j, task T_j is considered to have been invoked at time t. It is a property of the YARTOS programming model that the invocation of task T_j "occurring" at time t cannot have a deadline before time $t + p$. Therefore, during the interval $[t, t+p]$, T_j will not preempt T_i and when T_j is dispatched, there will be a message from task T_i for it to process.

The one exception to these invocation rules is when messages arrive from the outside world (*e.g.*, from interrupt handlers). When a task receives a message from an external process, the task's deadline is computed from the arrival time of the message (using application specified parameters that are sufficient for providing the desired real-time response to the external process).

With this message passing scheme, the time required in the worst case for a message to pass through tasks T_i and T_j in YARTOS is the same as the time required in the worst case for a message to be processed by a single task

that combined the functions of T_i and T_j. Thus the worst case end-to-end latency of a multi-threaded YARTOS application is not a function of the task structure. Rather, it is a function of the deadlines associated with application messages.

For example, in our videoconferencing application, all messages have a deadline for processing of 33 ms. If the *FG* task receives a message (an interrupt in this case) at time *t*, then if this message results in messages being sent to tasks *UP*, *CP*, *FF*, and *NI*, all messages will be processed at or before time $t + 33$. Therefore, each video frame is delivered to the network interface no more than 33 ms after it was generated.

The alternate approach to minimizing end-to-end latency is to combine all video processing tasks into a single task. However, in our system tasks *FF* and *NI* are actually general purpose operating system services that are shared with other user applications and hence can not be embedded directly into the conferencing application. (In fact, it is largely for this reason that a common approach to achieving real-time performance in general purpose operating systems has been to move application code into the operating system where finer-grain control over resource allocation is also usually possible.)

4. Related Work

Our message passing system is related to the paradigm of communication and scheduling integration reported by Draves *et al.* [2]. In this work a scheduling and context-switching mechanism based on the programming language concept of continuations is introduced to allow an applications that consists of multiple threads to execute more like a single threaded application. The emphasis is [2], however, was on reducing system overhead. Our work seeks to minimize worst case end-to-end latency.

Other related work includes the general priority model of Harbour *et al.* [3], wherein periodic tasks can be decomposed into subtasks that may have varying execution priority. In such a model it is possible to more directly express and reason about what we have called end-to-end latency constraints. In our work we have argued that a simple scheduling algorithm (described in [6]) is sufficient for managing latency.

Lastly, the flow shop scheduling results of Bettati and Liu [1] are relevant. They consider the problem of minimizing end-to-end latency in a system of multiple processing elements (*e.g.*, a distributed system). We have only considered the latency problem on a single shared processor.

5. Conclusions and Future Work

The design of general purpose operating systems impose constraints on the way one can structure real-time applications. Common operating system services such as network transport protocols, and device management need to be used by real-time applications. Because such services are shared with other applications they cannot by tightly bound to the real-time applications. We have shown that making application inter-task communication paths known to the kernel, one can provide a worst case end-to-end application latency bound that is the equivalent to the bound for an implementation of the application as a single task.

While we described the real-time message passing service within the context of an application whose tasks form a pipeline, the service can be applied to any graph structure to minimize latency of message communication along any path in the graph.

Currently we are porting the YARTOS message passing service to RT Mach (MK83) kernel and hope to compare the end-to-end latency of applications using the RT Mach and YARTOS communication primitives.

6. References

[1] Bettati, R., Liu, J.W.-S., *End-to-End Scheduling to Meet Deadlines in Distributed Systems*, Proc. IJDCS '92, Yokohama, Japan, pp. 452-459.

[2] Draves, R.P., Bershad, B.N., Rashid, R.F., Dean, R.W., *Using Continuations to Implement Thread Management and Communication in Operating Systems*, Proc. 13th ACM Symp. on Operating System Principles, Pacific Grove, CA, October 1991, pp. 122-136.

[3] Harbour, M.G., Klein, M.H., Lehoczky, J., *Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority*, Proc. 12th IEEE Real-Time

Systems Symp., San Antonio, TX, December 1991, pp. 116-128.

[4] Jeffay, K., Stone, D.L., and Smith, F.D., *Transport and Display Mechanisms for Multimedia Conferencing Across Packet-Switched Networks*, Computer Networks and ISDN Systems, to appear.

[5] Jeffay, K., Stone, D.L., and Smith, F.D., 1992. *Kernel Support for Live Digital Audio and Video.* Computer Communications, Vol. 16, No. 6 (July), pp. 388-395.

[6] Jeffay, K., 1992. *Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems*, Proc. 13th IEEE Real-Time Systems Symp., Phoenix, AZ, December 1992, pp. 89-99.

[7] Jeffay, K., Stone, D.L., Poirier, D., *YARTOS: Kernel support for efficient, predictable real-time systems*, in "Real-Time Programming," W. Halang and K. Ramamritham, eds., Pergamon Press, Oxford, UK, 1992. pp. 7-12.

[

8] Jeffay, K., *Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems*, Proc. 13th IEEE Real-Time Systems Symp., Phoenix, AZ, December 1992, pp. 89-99.

[9] Jeffay, K., *The Real-Time Producer/ Consumer Paradigm: A paradigm for the construction of efficient, predictable real-time systems*, Proc. 1993 ACM/SIGAPP Symposium on Applied Computing, Indianapolis, IN, ACM Press, February 1993, pp. 796-804.

[10] Liu, C.L., Layland, J.W., *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journal of the ACM, Vol. 20, No. 1, (January 1973), pp. 46-61.

[11] Tokuda, H., Nakajima, T., Rao, P., *Real-Time Mach: Towards a Predictable Real-Time System*, Proc.USENIX Mach Workshop, Burlington, VT, October 1990, pp. 73-82.

An Argument for a Runtime Layer in SPARTA Design

Robert W. Wisniewski and Christopher M. Brown
bob and brown@cs.rochester.edu
Department of Computer Science
University of Rochester
Rochester, NY 14627-0226

Abstract

Researchers have used advances in hardware technology to design larger and more complex real-time applications. Larger applications require new integration techniques while more complex applications require a restructuring of the underlying system support. We examine the system design issues of supporting SPARTAs (Soft PArallel Real-Time Applications). There exists a gap between hard real-time kernel mechanisms and the functionality desired by a SPARTA programmer. Thus, an integral part of supporting SPARTA design will be providing an intermediate runtime layer. We describe our experiences building Ephor, including what motivated its conception and development, and the resulting separation of responsibilities both easing SPARTA design and improving their performance.

1 Introduction

Many real-world applications contain both hard and soft real-time components. There has been considerable work on hard real-time system design such as [3] as well as work for parallel [1] and distributed [2] environments. Target applications for hard real-time systems include airplane autopilot and nuclear power plant control. New complex, parallel soft real-time applications have been generating considerable interest. Some example applications are: autonomous navigation, reconnaissance, and surveillance; operator-in-the-loop simulation; and teams of autonomous cooperating vehicles. The real-time community has acknowledged the need to explore issues raised by SPARTAs. Stankovic [4] enumerates a number of issues that we are directly addressing in the design of Ephor*. Among them are "what are the correct interfaces to robotics, RTAI, Vision, ... etc." and "what functionality should be in the OS level and what in the application level."

Designing a SPARTA is challenging, since in its full generality it calls for dynamic decision making about resource allocation, scheduling, choice of methods, and handling reflexive or reactive behavior smoothly within a context of planned or intended actions, and a host of other issues not typically encountered either in off-line or hard real-time applications. SPARTAs need different system support than either the large data-crunching scientific programs or the smaller less-structured applications currently being investigated in parallel environments. Further, supporting such applications was beyond the intended scope of previous real-time kernels because other more fundamental or lower level issues needed to be addressed first.

Hard real-time systems have not been designed to support the newly evolving soft real-time applications. In particular, they lack the flexibility needed to adjust to a complex and dynamic environment. The reason is that they must provide absolute predictability and guaranteed scheduling. Our runtime, Ephor, interacts with SPARTAs, maintaining hard real-time behavior when needed while providing graceful degradation in cases where performance is important but not critical to the success of the application. Our intermediate runtime layer is built on a hard real-time substrate providing the additional functionality needed by SPARTAs. The runtime reduces the replicated work of system monitoring and dynamic decision-making that is common between applications.

Initially, the effort needed to develop a general runtime package, such as Ephor, versus simply incorporating the needed portions into an application, may appear prohibitive. However, an analogy to threads

This material is based upon work supported by NSF Research Grant number CDA-8822724, DARPA Research Grant MDA972-92-J-1012, and ONR Research Grant number N00014-93-1-0221. The Government has certain rights in this material.

*Ephor was the name of the council of five in ancient Greece that effectively ran Sparta

0-8186-5710-3/94 $3.00 © 1994 IEEE

of control indicates this may not be so. Historically, threads of control were thought of simply as support for co-routining under direct user control. However, through time, many other issues with thread management have arisen. A similar situation applies with tasks, methods, or even planners in SPARTAs. Synchronization between tasks may be a significant issue, as might be the interleaving of tasks, or running one based on an exception generated by another, etc. Although it is conceivable these issues could be handled at the application level much the same way parallel thread management could be, there are compelling reasons for studying the systems aspects of such general capabilities.

The rest of the paper is structured as follows. Section 2 describes the motivation for, and our experiences leading to, designing a runtime. Section 3 contains the separation of responsibilities between the application, runtime, and kernel, facilitating SPARTA design. We present brief concluding remarks in section 4.

2 Motivating Factors

Problem Domain

Our research work in this area developed from a desire to study techniques for handling overdemand in real-time applications. Originally, our plan was to design a real-time application containing controllable overdemand situations. The shepherding problem we devised contains overdemand as well as many other general properties described below in **SPARTA Properties**. For a complete description of shepherding see [5]. Briefly, *sheep* (small vehicles) move around in a *field* (table) and a *shepherd* (robot arm) tries to maximize the number contained in the field. Before and during implementation we observed that there were several mechanisms that would have been useful but were not provided by current real-time systems such as concurrency control, dynamic technique selection, help in priority assignment, etc. As Stankovic [4] notes, "Because of these reasons many researchers believe that current kernel features provide no direct support for solving difficult time problems, and would rather see more sophisticated kernels..." A system that provided these features would be very useful. Rather than including these features into the kernel and sacrificing kernel predictability, we placed the additional functionality into Ephor, our runtime environment. This allows the designer to use the real-time kernel most appropriate for their environment. The application still receives the same functionality (actually more) and we maintain a predictable kernel.

SPARTA Properties

In designing Ephor in conjunction with the shepherding application we wanted to ensure the mechanisms developed for shepherding would be applicable to other programs. To do so, we designed the shepherding application to have many of the same properties as the soft real-time applications mentioned in the introduction. These applications contain an element of search whereby the agent determines the next course of action. Most are designed around a high-level executive instructing lower levels. The executive reasons using a model of the real world and carries out actions in it. The world is governed by general principles, but is not predictable. There is often an intermediate layer responsible for small corrections to the requested action (servoing). Also, there is often a low-level layer whose actions need to be carried out constantly and can occur "subconsciously", i.e., without intervention from the higher levels.

SPARTA Design Problems

Under the standard taxonomy there is only an application and kernel. Any operations or functions not performed by the kernel are the responsibility of the application. While the application needs to respond to the environment, determine the next course of action, and evaluate current progress, to perform reasonably it also needs to:

1. Monitor the load of the underlying processors. Often the application has a choice of tasks to accomplish the same end. Rather than submit impractical tasks, if the application knew the amount of compute power available, it could quickly choose the appropriate task to submit.
2. Maintain a list of allocated resources. As in 1, a wiser task submission based on task resource allocation yields better behavior.
3. Track execution times of tasks, especially highly variable ones. We have found that locally, the execution time of a task is fairly predictable, so knowing the last several execution times is useful.
4. Determine the correct interleaving of prioritized tasks.
5. Monitor other resources such as memory usage, bus utilization, etc.

It was our objective in designing Ephor to ensure the enumerated ideas could be handled by the generalized runtime, thus removing a significant burden from the SPARTA programmer.

Implementation Conclusions

We have implemented a real-time shepherding simulator and have almost completed implementation of the real-world shepherding application using small vehicles, a robot arm, and cameras. Ephor has been designed and implemented and handles a subset of the items enumerated in the previous section. Results using Ephor's mechanisms [5] indicate the potential of an intermediate runtime to facilitate the design of SPARTAs.

Designing the shepherding application led us to the conclusion that there is considerable functionality above the intended realm of real-time kernels that an increasingly large class of real-time applications strongly desire. It is best not to remove the predictability of the kernel since many critical applications depend on this property. However, there is a large class of applications willing to relax the tight constraints to gain increased functionality. These soft real-time applications will sacrifice predictability with or without a runtime. Further, if Ephor's mechanisms are not desired for a portion of the application they may be ignored causing no overhead to that portion. For this reason, and others mentioned in [5], a runtime layer will have a positive effect on SPARTA design. The only penalty if Ephor is completely ignored (or used only very minimally) is the one processor normally reserved to run it will be unavailable for other tasks. Ignoring Ephor though, defeats the purpose of the layering provided by the runtime. While it is currently possible to do so, we may discover through more experimentation and feedback that it will be best to prevent this. In the next section we describe the responsibilities of each layer.

3 Layer Responsibilities

By defining the responsibilities of each layer, we clearly define the obligations of the SPARTA designer. More importantly, we can specify the interface to the runtime, treating it as a "black box" with respect to the SPARTA programmer. A contribution and important part of our work is a clean separation of responsibilities for each layer and a description of mechanisms provided by Ephor independent of how they are coded. Thus, we not only have provided mechanisms, but a methodology.

Figure 1 shows the three layers in designing a SPARTA and underlying system. Information is shared across the runtime-application boundary by the data structure appearing in the Appendix. The data structure provides flexibility even beyond its primary purpose. For example, it is two-way writable so the application can give hints to the runtime by writing into fields the runtime is normally responsible for updating. The clean breakdown of information communication in Fig. 2 has been achieved by dividing layer responsibilities as detailed below.

Application

The application layer is solely responsible for responding to the environment. The application is responsible for communicating to the runtime (see Appendix) the different goals it will run throughout its execution, the different techniques it has for solving the goals, and the relative benefit of each technique. The application is responsible for determining the interaction of the environment and goals to produce the intended behavior. The application is responsible for indicating when a new goal needs to be solved in response to an environmental stimulus, or as the result of a previously completed goal. In essence the application must provide the flow of control to produce the desired program.

Runtime

The runtime receives the structure of the goals the application will submit throughout its execution via the data structure in the Appendix. As can be observed from the Appendix, items implement techniques, and techniques satisfy goals. The runtime is responsible for determining the execution time of the different techniques for solving the goals. The application can explicitly provide the times, or the runtime may dynamically gather them during the program's execution. The latter method allows Ephor to adapt in a changing environment. Ephor is responsible for determining and running the appropriate items needed for completing a selected technique. In selecting the technique to solve a requested goal, Ephor need not only be aware of the program structure and application, but also of the internal state of the system. The runtime is therefore responsible for interfacing to the underlying kernel to obtain the information it needs. It is responsible for monitoring the following resources necessary to select dynamically the best technique: processor load, expected available processors for running parallel techniques, memory or cache utilization, bus utilization, and other current resource allocation such as range sensors, manipulator, or cameras. Ephor is responsible for maintaining a central location where resource allocation information can be quickly and coherently obtained by either Ephor or the application.

To provide a comprehensive runtime package we have implemented many mechanisms in Ephor. While

Real-World Layer	Responsibilities	Layer Division	Characteristics
Application Layer	Respond to environment Generate goals Provide goal solving structure	Executive Level	agent determines next action, search executive instructs lower levels
		Intermediate Level	corrections to actions(servoing) interpret high level
		Lowest Level	survival actions, sensing, manipulating
Runtime Layer (Ephor)	Remain domain independent Use program structure representation Provide mechanisms to the application Monitor the underlying system	Interface to Application	shared data structure, mechanisms, system information to application
		Interface to real-time substrate	monitor underlying system, schedule tasks
Real-Time Substrate	Provide basic real-time properties Make more information available (e.g. resource allocation)		

Figure 1: The Three Layers in the Design of a Real-World Application

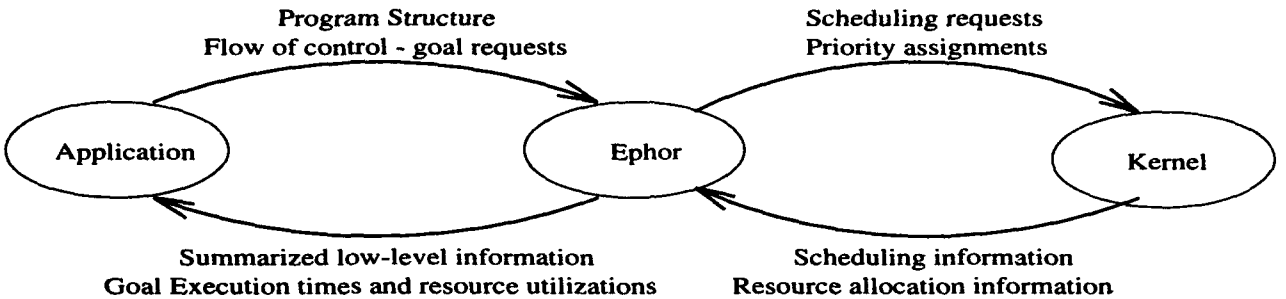


Figure 2: Layer Information Exchange

it is beyond the scope of this paper to discuss them in detail individually, we give a list to provide an overview of Ephor's functionality: 1) dynamic technique selection based on internal system state, 2) schedule tasks using derivative worst case policy, 3) dynamic parallel process control, 4) de-scheduling of all running tasks associated with a particular goal, 5) automatically time tasks and update their status block, 6) data structure to share information between the runtime and application, 7) parallel scheduling of hard and soft real-time tasks, 8) overdemand detection and recovery*, 9) early termination of tasks*, 10) automatic resource allocation based on goal priority*.

A difficulty in soft real-time systems is evaluating the performance of a given mechanism since there is often not a hard metric that can be used to judge success or failure. Part of our continuing work involves defining suitable metrics for measuring our mechanisms. Here we point to one particular case of how the dynamic technique selection mechanism of Ephor performs (a more complete analysis of this mechanism can be found in [5]). Figure 3 represents the performance of two different planners (A and B) and the dynamic selection of them (with runtime). The runtime results were obtained by having Ephor dynamically select between the two planners based on the internal state of the system.

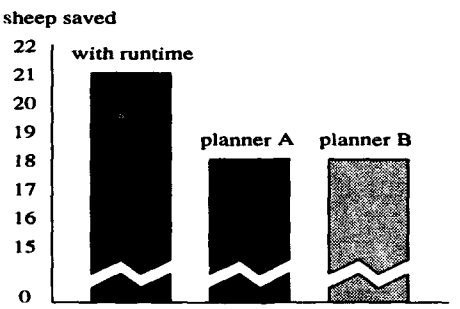


Figure 3: Performance of dynamic technique selection

Kernel

We assume a hard real-time substrate. The kernel is expected to provide fundamental real-time properties and mechanisms such as a predictable scheduling policy, an accurate real-time clock, guaranteed deadlines, task priorities, and other properties typically associated with real-time kernels [4]. The kernel is responsible for handling interrupts, and allocating resources as directed by Ephor.

4 Conclusions

In this paper we discussed the difficulties in SPARTA design and argued for a runtime layer to alleviate some of these problems. The benefits of our

*planned but not yet implemented

run-time system are two-fold. First, there are software engineering and programming benefits: application programmers can specify their needs without knowing how they will be fulfilled; and a common set of mechanisms can be reused across many domains without recoding. Second, by thoroughly investigating many possibilities, we can include in the runtime those mechanisms yielding the best performance.

In designing a Soft PArallel Real-Time Application (SPARTA) we discovered that the functionality provided by current hard real-time operating systems was limited. We developed Ephor, a runtime environment, to increase the functionality of the system while maintaining a predictable kernel. Our design provides a clean separation of responsibilities, facilitating SPARTA design and improving their performance.

Acknowledgements

Tom LeBlanc provided helpful discussions and comments on this work.

References

[1] V. P. Holmes and D. L. Harris. A designer's perspective of the hawk multiprocessor operating system kernel. *Operating Systems Review*, 23(3):158–172, July 1989.

[2] K. Schwan, P. Gopinath, and W. Bo. Chaos-kernel support for objects in the real-time domain. *IEEE Transactions on Computers*, 36(8):904–916, August 1987.

[3] J. Stankovic and K. Ramamritham. The design of the spring kernel. *Proceedings of the Real-Time Systems Symposium*, pages 146–157, December 1987.

[4] John A. Stankovic. Real-time operating systems: What's wrong with today's systems and research issues. *Real-Time Systems Newsletter*, 8(1):1–9, 1992.

[5] Robert W. Wisniewski and Christopher M. Brown. Ephor, a run-time environment for parallel intelligent applications. In *Proceedings of The IEEE Workshop on Parallel and Distributed Real-Time Systems*, pages 51–60, Newport Beach, California, April 13-15, 1993.

Appendix

Ephor-Application Interface (abridged)

```
struct item_t {
  int imp_kind;        /* indicates if a function or technique implements item */
  union implementer_t{
    funct_t f_implement;  /* pointer to function that implements this item */
    struct technique_t *t_implement; /* pointer to the technique implementing item */
  } implementer;
  int complete; /* 0 means nothing, 1 means done, anything else is left to
                   the interpretation of particular item */
};
struct technique_t {
  int priority;  /* 0 -> n the lower the number the higher the priority */
  int cpu_time;  /* the median of the last three times, provides a good
                    dynamic estimate rather that using worst case time */
  int cpu_times[3];  /* used to compute cpu_time see above */
  int cpu_index; /* indicate which cpu_time index to write to */
  float cpu_per; /* percentage of an INTERVAL of a cpu this technique needs */
  float max_change;  /* number indicating the maximum percentage change */
  int cutoff_time;  /* when the system should terminate this technique */
  int memory;    /* space required by this technique, instructions and data */
  int *item_par_des;  /* the parallel ordering on items similar numbers may proceed
                         in parallel, low numbers must be run before higher numbers */
  struct item_t *item; /* set of items that implement this technique */
};
struct goal_t {
  boolean periodic; /* TRUE indicates task is periodic */
  int rate;  /* if periodic is true the system runs this goal at rate */
  int run_technique;  /* technique that should currently be run to satisfy goal */
  int numb_techniques;  /* number of different possible techniques for this goal */
  struct technique_t technique[MAX_TECHS]; /* list of techniques to solve goal */
};
MAIN struct goal_t goal_list[MAX_GOALS];/* a list of goals for the application*/
```

Real-Time Platforms and Environments for Time Constrained Flexible Manufacturing*

John A. Stankovic, Krithi Ramamritham and Goran Zlokapa
Department of Computer Science
University of Massachusetts
Amherst, Mass 01003

Abstract

The Spring Kernel and associated algorithms, languages, and tools provide system support for static or dynamic real-time applications that require predictable operation. Spring currently consists of two major parts: (1) the development environment, where application and target systems are described, preprocessed and downloaded, and (2) the run-time environment, where the operating system, the Spring Kernel, creates and ensures predictable executions of application tasks. Very recently, we have integrated our real-time systems technology with component technologies from robotics, computer vision, and real-time artificial intelligence, to develop a test platform for flexible manufacturing. But the results being produced are generic so that they should be in many other real-time applications such as air traffic control and chemical plants. In this paper we describe this platform, identify new features that we developed, and comment on some lessons learned to date from this experiment.

1 Introduction

The Spring Kernel [6] is designed to conform to the requirements of a wide range of highly predictable and dynamic real-time applications. The strength of its run-time support lies in the use of its two distinct scheduling concepts as well as its support for both predictability and flexibility. In regards to scheduling, it supports a once-guaranteed-always-guaranteed policy suited for applications, or parts of an application that impose stringent time and resource requirements, or where once an operation begins it cannot be undone. It also provides a best-effort type of scheduling, where the system dynamically creates a schedule to maximize the total accrued system value. Subject to system requirements, these two scheduling concepts can coexist within one application. The kernel also retains a significant amount of semantic information at runtime, supplied by the system designers and programmers as well as extracted by various tools such as the compiler. This information is then utilized at runtime providing a high degree of flexibility when required. This flexibility is subject to the scheduling rules and algorithms so that predictability is retained even as the system is adapting. We refer to this architectural strategy as a reflective architecture [7]. While many papers have been written describing various innovations in the Spring kernel and its associated scheduling algorithms [5, 6], in this paper we report on new extensions and lessons learned when applying the ideas and system to flexible manufacturing [1, 2]. The extensions arise both in developing a complete runtime platform and because of the need for integrating component technologies from robotics, computer vision, and real-time AI with this real-time computing platform.

2 The Flexible Manufacturing Testbed

The basic idea underlying the testbed is to model applications which provide predictable responses while functioning in nondeterministic environments. In this testbed (see Figure 1), objects or material needed by consuming processes are introduced nondeterministically into a circular queue awaiting processing. A consuming process can request (a specific combination of) objects at any time, and it can set arbitrary deadlines and values for the objects needed by it. The circular queue can control the type of incoming material and the rate at which it is processed.

To maximize total accrued system value (in terms of objects delivered in time to consuming processes) and resource utilization, Spring's dynamic scheduling

*This work is funded by the National Science Foundation under grant IRI-9208920.

0-8186-5710-3/94 $3.00 © 1994 IEEE

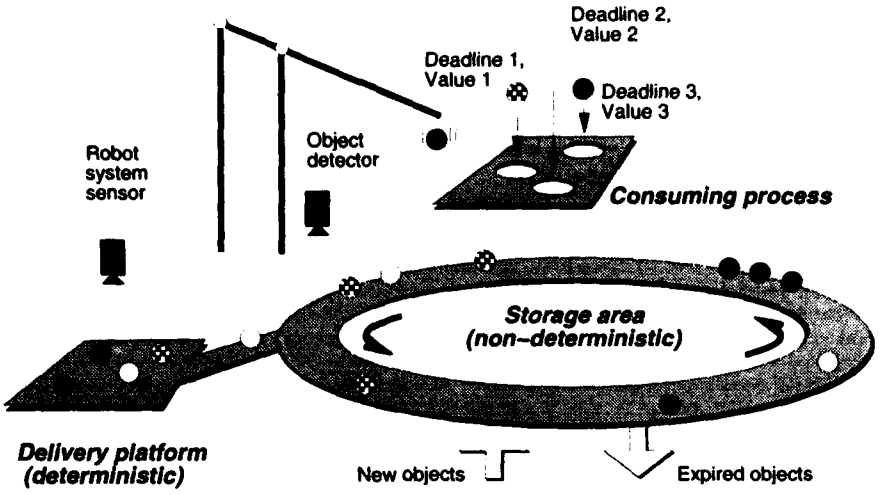


Figure 1: Schematic of the Flexible Manufacturing Testbed

algorithm is used to manage the resources in the system. Specifically, based on resource availability, specific requests of consuming processes, as well as specific timing constraints, objects are removed from the circular queue and placed on a delivery platform in a timely manner. The delivery platform represents a user specified, finite capacity intermediate storage. All objects on this platform are guaranteed to meet their timing constraints and to be delivered to the downstream processes as requested and in time.

The entire specification of such a system, from application processes to target hardware, is described using a System Description Language (SDL) [4]. On the other hand, all system dynamics are handled by the Spring Kernel.

This testbed has characteristics applicable to a wide variety of actual, real world applications including:

1. Air Traffic Control — airplanes can enter a holding pattern, they can leave if fuel is low or if they cannot be guaranteed timely landing, required terminal, and service crew. If they are admitted to the runway they must be guaranteed full service.

2. Chemical Plants — different chemicals ready for mixing are placed on the conveyer belt. The consuming processes require chemicals to be mixed in a certain order. Due to chemical deterioration, each chemical solution must be either delivered within a certain time, or it must be disposed to prevent hasards.

3. Flexible Manufacturing — robotic workcells, depicted in the figure above as the consuming process, can perform incremental operations such as mechanical assembly or quality control inspection, and deliver finished components to a subsequent manufacturing operation.

3 Extensions and Lessons Learned

Our prior work on scheduling and operating system support for real-time systems [5, 6] as well as recent work on software description languages and tools [4] was not targeted at any particular application. Our goal was to design them to be applicable to time-critical applications that had a broad set of characteristics.

In this section we discuss how some of these concepts and their practical realizations in Spring had to be extended to apply them to the flexible manufacturing platform.

3.1 Scheduling Extensions

For flexible manufacturing, it was necessary to support tasks with precedence constraints, shared resources and multiprocessing. Spring scheduling algorithms had been designed with such requirements in mind [5].

However, additional requirements were imposed by the fact that scheduling must occur at different levels of abstraction. At the higher level the system must deal with orders for products, resources which consist of parts and subcomponents to be automatically assembled - constrained by robots, floor space, cost, and expected profits. Decisions made at this level, handled by the real-time AI subsystem, determined which of the incoming orders need to be carried out, computational resources permitting. Whether it was possible to carry out a specific order was determined by the scheduler at the lower level, where the system deals with the *computational resources* needed to move robots, assemble products, etc. It was neces-

sary to implement both levels of scheduling with a feedback interface between these levels. For example, if the higher level decides to make certain products, the actual manufacturing floor may not be capable of performing these tasks in time. Such information supplied to the higher level scheduler improves performance of the system in choosing between alternatives.

Interesting research questions regarding multiple levels of scheduling include:

- how to partition the scheduling functionality between the high level (RTAI) planner and the system level scheduler (which is also a planner),
- what information should be passed back and forth between the levels, and
- how to pass information back and forth between the schedulers so as to get the best performance.

In developing the flexible manufacturing testbed we also found it necessary to be able to hold resources across a set of processes. For example, a set of processes may require a common tool which cannot be shared with others. This adds an interesting scheduling complication not typically addressed by real-time scheduling algorithms. We also found that deadlines can sometimes be relaxed within a *deadline tolerance* [3]. Interesting questions involve understanding the cumulative effect of missing the original deadline, but satisfying the tolerance factor.

3.2 Extensions to the Kernel to Interface with External Subsystems

As originally implemented, the Spring kernel controlled computational activities running on processors within the multiprocessor Spring node(s). However, in the flexible manufacturing platform there was a need to control, in a timely fashion, not only these activities, but also robotic and vision processes running outside the Spring node(s). This is because, given that commercial robotics and vision subsystems are highly developed and efficient, it is necessary to be able to link such subsystems into a flexible manufacturing plant without re-implementation. To do this, we defined and implemented a remote interface so that such subsystems can be added to the Spring runtime platform, as long as they adhere to the interface requirements. The interface allows the Spring scheduler to control the timing properties of the external subsystems. The Spring, vision, and robotic subsystems interact via distributed shared memory, implemented using ScramNet[8].

3.3 System Description Language

The system description language (SDL) had been designed to provide a way for developers to specify the properties of all parts of the system in great detail, independent of the application functionality [4]. To provide easier system integration and to assist in on-line scheduling, a method of compiling or mapping program representations to task representations was also developed. The compiled information is available for use by other tools as well as the runtime system that use, modify, or add to it. In short, SDL provides support for specification, compilation, and execution of applications on the Spring system. It is also used for the specification of simulations run under Spring's scheduling testbed. In this way, both the actual system and the simulations complement each other. This is especially important in robotic applications where testing must be done with simulated robots for safety reasons, before you control the actual robots.

While the SDL provides software support for specifying important information, extensions were necessary when used with the flexible manufacturing platform:

- techniques to describe the interfaces with the robotic and vision subsystems and how they are expected to operate (including the expected workloads),
- specification of fault tolerance requirements and design, and
- linking computer aided design (CAD) and design for manufacture (DFM) tools to SDL.

Another key issue is the need for tools to aid the system builder in deriving deadlines from time, value and fault properties of the application and environment. In flexible manufacturing there is a combination of loose time constraints and very precise time constraints which are linked to each other. For example, the arrival of parts, orders for products, and delivery dates and times usually have loose and unpredictable timing requirements. Yet, sometimes delivery times are very strict and missing the deadline can cause serious financial loss. Further, once a product is begun there are often tight and strict deadlines to be met in the form of commands to the robots else the product is ruined, or worse the robot itself is damaged.

3.4 Integrated Simulation Support

Simulation is critical to flexible manufacturing. We are developing a simulator that is integrated in three ways: across multiple levels of scheduling, with the runtime system model, and with actual robotic workcells.

In order to understand the operation of the system prior to putting it into use, we have developed a simulator that is driven from an application level view. This consists of being able to order products, have raw materials arrive, and see the flexible manufacturing floor in operation, i.e., visualize the robots assembling products. Then it is possible to *zoom* into

the system level and see what computational entities are actually running to effect the application level actions. The schedules are dynamically changed as the higher level actions proceed, precedence constraints can be displayed, and summary statistics continuously updated. These features provide the integration across levels of scheduling.

The simulator also accurately follows the runtime model of the flexible manufacturing testbed system so that the same workloads, algorithms, etc. are operating in both systems. The exact same tests can drive both systems and the outputs can be directly compared.

We also have plans to integrate n copies of the simulator running on workstations with the actual robot workcell so that a distributed factory floor can be modeled. This will permit experimentation with distributed systems and coordination algorithms.

4 Summary

Attempting to use the Spring algorithms and kernel to implement the flexible manufacturing platform has resulted in significant improvements in the system and has identified a number of new research issues some of which have been briefly discussed here. As of this writing the physical testbed has been completely implemented, the kernel, SDL and simulator are all functioning with two exceptions: (i) part of the interface between the Spring scheduling algorithm and the real-time AI planner remains to be fully debugged, and (ii) minor modifications required by the scheduling algorithm must be made. These should be completed shortly.

References

[1] R. Ayres and D. Butcher, The Flexible Factory Revisited, *American Scientist*, Vol. 81, September 1993.

[2] A. Baker and M. Merchant, Automatic Factories, *IEEE Spectrum*, December 1993.

[3] G. Buttazzo and J. Stankovic, RED: A Robust Earliest Deadline Scheduling Algorithm, *Third International Workshop on Responsive Computer Systems*, Sept. 1993.

[4] D. Niehaus, J. Stankovic, and K. Ramamritham, The Spring System Description Language, Univ. Of Massachusetts Technical Report, TR 93-1, January 1993.

[5] K. Ramamritham, and J. Stankovic, Scheduling Strategies Adopted in Spring: An Overview, *Foundations of Real-Time Computing: Scheduling and Resource Management*, Andre van Tilborg and Gary Koob, Ed., Kluwer Academic Publishers, pp. 277-305, 1992.

[6] J. Stankovic and K. Ramamritham, The Spring Kernel: A New Paradigm for Real-Time Systems, *IEEE Software*, Vol. 8, No. 3, pp. 62-72, May 1991.

[7] J. Stankovic and K. Ramamritham, A Reflective Architecture for Real-Time Operating Systems, in *Principles of Real-Time Systems*, Prentice Hall, to appear 1994.

[8] SYSTRAN Corporation, Scramnet Network Reference Manual, Dayton, Ohio, 45432.

Session VII: Concurrency Control

Chair: Vic Wolfe
U. Rhode Island

A Mixed Locking/Abort Protocol for Hard Real-Time Systems

LihChyun Shu Michal Young
{shu,young}@cs.purdue.edu
Purdue University
Department of Computer Sciences
West Lafayette, IN 47907-1398

1 Introduction

Serializability greatly simplifies reasoning about correctness in concurrent systems, including real-time systems. Our research addresses concurrency control protocols that accommodate analytic guarantees of schedulability, can be implemented with small bounded overheads and blocking, and ensure serializable execution of entire tasks including complete read/compute/write cycles (as opposed to serializable execution only of short embedded transactions without computation.)

One such protocol which combines locking and abort is described below. Among its interesting properties are that transactions scheduled by locking are never aborted, tasks are aborted only due to conflict with higher priority tasks, and the cost of abortion can be bounded for the purpose of schedulability analysis.[1] The protocol is illustrated with an avionics example adapted from [4]. The priority ceiling protocol can ensure schedulability of 8 tasks if serializability of only short sequences of data accesses is required, but cannot schedule even the first 2 tasks if serializability is required for whole tasks. Under reasonable assumptions our protocol achieves schedulability of the first 6 tasks while guaranteeing serializability of entire tasks.

2 Model and assumptions

We consider schedulability for a set of tasks, $\{\tau_1, \tau_2, \ldots, \tau_n\}$. We consider each task also as a transaction which may contain read and write operations to shared data and is a unit of consistency, i.e., if executed alone, it transforms the shared data from a consistent state to another consistent state. T_i is the transaction associated with task τ_i, $1 \leq i \leq n$. $r_i[x]$ and $w_j[x]$ denote T_i's read and write operation on data item x, respectively. Two operations are said to conflict if they both operate on the same data item and at least one of them is a write.

[1] Contrast to the abort-oriented protocol presented in [5], which completely avoids priority inversion but does not bound abortion cost.

$<_i$ denotes the execution order of operations in each transaction T_i. A schedule is a partial order $(H, <_H)$ such that (1) $H = \cup_{T_i \in T} T_i$; (2) $<_H \supseteq \cup_{T_i \in T} <_i$ and (3) for any two conflicting operation $p, q \in H$, either $p <_H q$ or $q <_H p$.

We assume static priority assignment based on rate monotonic analysis (RMA) [3]. P_i denotes the period of τ_i. To control priority inversion, we use a variant of the the stack resource policy (SRP) [1], treating reads and writes differently as in the read/write priority ceiling protocol (PCP) [6]; we will call this R/W SRP. It is interesting to note that the stack-based discipline is critical to the correctness of the mixed protocol described below, as well as its performance.

3 Mixed locking/abort protocol

SRP and read/write PCP are pure locking protocols with the characteristic that worst-case blocking is limited to the sizes of embedded transactions or critical sections. In [6], PCP is termed a 2-phase locking protocol, but embedded transactions are presumed to contain only database operations. This limitation makes controlling priority inversion easier, but reasoning about overall system correctness harder. Unfortunately, a 2PL-scheduled transaction consisting of a read-compute-write sequence must not begin to release locks until the last lock has been obtained; critical sections may therefore be nested and may extend across the middle compute steps.

The protocol described below uses R/W SRP as a baseline strategy, but schedules a transaction T by abort when schedulability analysis reveals that T causes excessive blocking. T immediately releases each read lock after its read access, hence incurring less blocking to higher-priority transactions. Since the protocol is not 2-phase, a combination of locking and timestamps are used to order conflicting operations.

We assume a source of monotonically increasing timestamps. Each data item x is associated with two timestamps: Rcur[x] from the most recent read operation and Wmax[x] from the most recent write operation.

0-8186-5710-3/94 $3.00 © 1994 IEEE

- In transactions scheduled by pure locking, locks are requested as in 2PL. When a transaction T_i commits, it obtains its timestamp ts(T_i) and for each datum x read(written) by T_i, sets Rcur[x] (Wmax[x]) equal to ts(T_i) and then releases its lock on x.
- In transactions scheduled by locking and abort, T_j obtains its timestamp ts(T_j) at initiation.

  For each item x accessed in the read phase, T_j first sets a read lock on x. It then checks whether Wmax[x] > ts(T_j) and aborts if so; otherwise it reads x. Rcur[x] is set to ts(T_j), and the read lock on x is released.

  For each item x accessed in the write phase, T_j sets a write lock on x. It then checks whether Rcur[x] > ts(T_j), and if so it aborts. If Wmax[x] > ts(T_j), T_j releases the write lock without writing (an application of Thomas' write rule [7]). Otherwise, it places x on wList and delays the actual write operation until the commit phase.

  In the commit phase, each x on wList is written, Wmax[x] is set to ts(T_j), and then the write lock on x is released.

As in read/write PCP, locks in our protocol are set and released by changing the r/w priority ceilings of the corresponding data items. The protocol is not two phase since read locks of each transaction scheduled by abort/locking are released immediately after access. To guarantee serializability, timestamps are utilized, in addition to locks, to order conflicting operations.

Observe that Wmax[x] holds the maximum timestamp of transactions that updated Wmax[x] (it increases monotonically) but Rcur[x] holds the timestamp of the transaction that most recently updated Rcur[x]. Permitting Rcur[x] to be updated in the "wrong" order complicates the correctness proof, but eliminates a costly critical section in an implementation.

4 Properties of the mixed protocol

We denote operations on x by $o[x]$, $p[x]$ and $q[x]$ where it is not necessary to distinguish between read and write. ts_k denotes acquisition of a timestamp by transaction T_k. $U_k^o[x]$ denotes T_k's updating of Rcur[x] if $o = r$ or Wmax[x] if $o = w$. We assume either $U_k^o[x] <_H U_l^o[x]$ or $U_l^o[x] <_H U_k^o[x]$ for any two transactions T_k and T_l and $o = r$ or w.

Observation 1 *For every transaction T_k and an operation $o_k[x]$ of T_k, $ts_k <_k U_k^o[x]$ and $o_k[x] <_k U_k^o[x]$. Hence, $ts_k <_H U_k^o[x]$ and $o_k[x] <_H U_k^o[x]$ for every schedule H.*

This follows from the rules that a transaction acquires its timestamp before it updates either Rcur or Wmax, and it performs each of its operations before it updates either Rcur or Wmax.

Definition 1 *Given any operation $q_j[x]$, we call $p_i[x]$ the immediate preceding conflicting operation of $q_j[x]$ if $p_i[x]$ executes before and conflicts with $q_j[x]$, and $\nexists o_k[x]$ such that $o_k[x]$ also conflicts with $q_j[x]$ and $U_i^p[x] <_H U_k^o[x] <_H U_j^q[x]$.*

Note that $o_k[x]$ may or may not conflict with $p_i[x]$.

Observation 2 *If $o_k[x]$ conflicts with $p_i[x]$, then $U_k^o[x] <_H U_i^p[x]$ iff $o_k[x] <_H p_i[x]$.*

Informally, timestamp order is consistent with the order of operations. This follows from the rules that each conflicting lock is not released until Rcur or Wmax has been updated. By Definition 1 and Observations 1 and 2, we have $U_i^p[x] <_H q_j[x] <_H U_j^q[x]$. The next observation follows directly from Definition 1.

Observation 3 *If $q = w$ (a write operation), then both Rcur[x] and Wmax[x] will remain unchanged after $U_i^p[x]$ is executed and before $q_j[x]$ is executed. If $q = r$, then Wmax[x] will remain unchanged after $U_i^p[x]$ is executed and before $q_j[x]$ is executed.*

Property 1 *The mixed locking/abort protocol guarantees serializability.*

Proof: Given any operation $q_j[x]$, let $p_i[x]$ be its immediate preceding conflicting operation. We first show if $q_j[x]$ was not rejected or ignored, then ts(T_i) < ts(T_j). There are four cases to consider:

Case I1 Both T_i and T_j are scheduled by pure locking: By the pure locking scheduling rule, the lock on x is not released until timestamp is acquired and properly attached. Since $p_i[x]$ precedes and conflicts with $q_j[x]$, T_i must obtain its timestamp before T_j did. Hence, ts(T_i) < ts(T_j).

Case I2 T_i is scheduled by abort/locking while T_j is scheduled by pure locking: T_i obtains its timestamp when it starts. Also, a lock is set on x before $p_i[x]$ is performed. Since T_j obtains its timestamp when it commits, ts(T_i) < ts(T_j).

Case I3 T_i is scheduled by pure locking while T_j is scheduled by abort/locking: Since both T_i and T_j set a lock on x and $p_i[x]$ precedes $q_j[x]$, by the time $q_j[x]$ is about to be performed $p_i[x]$ and $U_i^p[x]$ must have been done. Since $p_i[x]$ is the immediate

preceding conflicting operation of $q[x]$, by Observation 3, no conflicting operations will update Rcur[x] or Wmax[x] or both after $U_i^p[x]$ is executed and before $q_j[x]$ is executed. If ts(T_i) > ts(T_j), then T_j will be aborted unless $p = q = w$ (in that case $q_j[x]$ will be ignored by Thomas' write rule).

Case I4 Abort/locking used for both T_i and T_j: similar to case I3.

Now suppose ts(T_i) < ts(T_j). We show ts(T_k) < ts(T_j) for any other preceding conflicting operation $o_k[x]$ of $q_j[x]$. Suppose to the contrary that ts(T_k) > ts(T_j). We have $ts_i <_H ts_j <_H ts_k$ because ts(T_i) < ts(T_j) < ts(T_k). There are two cases to consider:

Case O1 $o_k[x]$ and $p_i[x]$ conflict: $o_k[x]$, $p_i[x]$, and $q_j[x]$ are pairwise conflicting operations. By our assumptions, $o_k[x] <_H q_j[x]$ and $p_i[x] <_H q_j[x]$. If $p_i[x] <_H o_k[x]$, then $p_i[x] <_H o_k[x] <_H q_j[x]$. By Observation 2, $U_i^p[x] <_H U_k^o[x] <_H U_j^q[x]$, which violates Definition 1 for immediate preceding conflicting operations. Hence, we must have $o_k[x] <_H p_i[x] <_H q_j[x]$ and $U_k^o[x] <_H U_i^p[x] <_H U_j^q[x]$. By Observation 1, $ts_k <_H U_k^o[x]$. Hence, the schedule must contain $ts_i <_H ts_j <_H ts_k <_H U_k^o[x] <_H U_i^p[x] <_H U_j^q[x]$. Because operations of T_i and T_j are not properly nested, this execution violates the stack-based discipline.

Case O2 $o = p = r$ and $q = w$: Both T_k and T_i must modify Rcur[x], i.e., $U_k^r[x]$ and $U_i^r[x]$. Since both $r_k[x]$ and $r_i[x]$ precede and conflict with $w_j[x]$, by Observation 2, $U_k^r[x] <_H U_j^w[x]$ and $U_i^r[x] <_H U_j^w[x]$. By our assumption, either $U_k^r[x] <_H U_i^r[x]$ or $U_i^r[x] <_H U_k^r[x]$. If $U_i^r[x] <_H U_k^r[x]$, then we have $U_i^r[x] <_H U_k^r[x] <_H U_j^w[x]$, which violate our assumption that $r_i[x]$ is the immediate preceding conflicting operation of $w_j[x]$. Hence, $U_k^r[x] <_H U_i^r[x]$. The schedule thus must contain $ts_i <_H ts_j <_H ts_k <_H U_k^r[x] <_H U_i^r[x] <_H U_j^w[x]$, which again violates the stack-based discipline. □

Property 2 *Transactions scheduled by pure locking will never be aborted.*

Proof: Follows from Cases I1 and I2 in the proof of Property 1. □

When a transaction is aborted, we must remove its effect to the database as well to other transactions. If the protocol were not designed carefully, this might trigger further abortions, often termed *cascading* abort.

Property 3 *Executions produced by the mixed protocol are cascadeless.*

Proof: Follows from the scheduling rules that write locks are held until after a transaction commits or aborts. □

Property 4 *If a transaction is aborted, then it is aborted by a higher-priority transaction.*

Proof: Suppose T_j must be aborted because its operation $q_j[x]$ can not be performed because a conflicting operation $p_i[x]$ of T_i has been executed and ts(T_i) > ts(T_j). We want to show that T_i's priority must be greater than T_j's. Since ts(T_i) > ts(T_j), $ts_j <_H ts_i$. By Observations 1 and 2, the schedule thus contains $ts_j <_H ts_i <_H U_i^p[x] <_H q_j[x]$. If T_j has higher priority than T_i, then T_j is blocked by lower-priority transaction T_i after T_j starts, which violates the stack-based discipline. Hence, T_i's priority must be higher than T_j's. □

Observation 4 *When transactions T_i and T_j contain two conflicting operations $p_i[x]$ and $q_j[x]$, respectively and T_i's priority is higher than T_j's, then the worst-case abortion cost for T_j by T_i due to conflict on x is the longest execution time from T_j's initiation up to (but not including) $q_j[x]$.*

This can be observed in the proof of Property 4: if $q_j[x]$ has been executed before T_i preempts T_j, then T_j will not be aborted by T_i due to conflict on x. We call such cost *prefix* abortion cost for T_j. If T_i and T_j conflict on more than one data item, we need only consider the largest such cost. We call such cost *A-cost*$_{i,j}$.

Notice when a transaction is aborted, due to delayed writes there is no need to undo its write operations. In addition, there is no need to undo its read operations, i.e., to recover the old value of Rcur for each data item read by the transaction. Those reads can be thought of as "ghost" operations and cause no harm to overall system correctness. As a result, the overheads in aborting a transaction are very small.

Lemma 1 *The worst-case abortion cost charged to task τ_j by conflicting task τ_i is $\lceil \frac{P_j}{P_i} \rceil \times$ A-cost$_{i,j}$.*

Proof: Because of stack-based discipline, each instance of τ_i can abort τ_j at most once. The Lemma thus follows from Observation 4. □

We define $HPC_j = \{\tau_i$: τ_i has higher priority than τ_j and τ_i conflicts with $\tau_j\}$.

Lemma 2 *The worst-case abortion cost for τ_j is $\sum_{\tau_i \in HPC_j} (\lceil \frac{P_j}{P_i} \rceil \times$ A-cost$_{i,j})$.*

Proof: Follows from Lemma 1 and Property 4. □

5 Application and schedulability analysis

When performing schedulability analysis, we first assume that all tasks are scheduled by pure locking. If all critical tasks are schedulable, there is no reason to consider scheduling tasks by abort/locking. Otherwise, we add timestamp management overheads and again calculate worst-cast execution times and blocking duration. The schedulability analysis will then determine which tasks must be scheduled by abort/locking.

Whenever a task τ_i is found not schedulable due to excessive blocking caused by a lower priority task τ_j, we schedule τ_j by abort. We calculate worst-case abortion cost for τ_j based on Lemma 2. This abortion cost effectively becomes part of the computation time for τ_j. We then re-evaluate worst-case blocking to τ_i and resume the schedulability analysis. Because the new blocking caused by τ_j cannot be greater than before, schedulability results for tasks with higher priority than τ_i will not be affected.

An example

The avionics platform example in [4] has 18 periodic tasks and 9 data objects. As in [4], task Weapon_Release is ordered second (not in pure rate-monotonic order) to meet a 5 ms jitter requirement.

We assume:[2]

- timestamp acquisition takes at most 50 μs.
- each update of Rcur or Wmax takes at most 0.5 μs.
- each read/write of a datum takes at most 0.5 μs.
- reading Rcur or Wmax and comparing it to a transaction's own timestamp takes at most 1 μs.
- set/release a lock takes at most 50 μs.
- for each datum, access wList takes at most 50 μs

Table 1 shows task set characteristics from [4] but with blocking calculated from execution times of whole tasks rather than short critical sections. The schedulability analysis is based on the "critical time test" reported in [4]. Task Weapon_Release is not schedulable in a pure locking protocol due to its stringent jitter requirement and excessive blocking by lower-priority tasks, even though cumulative task utilization is only 6.15%. To schedule Weapon_Release and meet its jitter requirement, we must permit abortion of each lower-priority task that reads "DB" and whose execution time is greater than 1. (If the jitter requirement is removed as in [2], then all the tasks will be schedulable by pure locking.) All other tasks can be scheduled by pure locking.

Table 2 illustrates the schedulability calculations under the new protocol with all overheads for the first six tasks.

[2] These assumptions are based on simple memory accesses for most operations, with no context switches or operating system services, and the protocol is designed to make such an implementation possible.

6 Conclusion

The non-2-phase protocol described above can improve system schedulability while maintaining a strong correctness criterion, i.e., serializability (of whole tasks and not just of small sequences of data accesses). It is an example of a concurrency control protocol suitable for hard-real-time systems, and is (to our knowledge) the first example of a protocol that may abort tasks and yet achieve better *worst case* schedulability than a pure locking protocol. Additional overheads are incurred for timestamp management and abortion, but these overheads are small and need not be incurred by all tasks. The choice of which tasks require timestamp management and possible abortion is straightforward and driven by schedulability analysis. The general avionics example adapted from [4] illustrates a class of system for which selective abort may increase the number of tasks whose worst-case schedulability can be guaranteed.

References

[1] T. Baker. Stack-based scheduling of realtime processes. *Journal of Real-Time Systems*, 3:67–99, 1991.

[2] T.-W. Kuo and A. K. Mok. SSP: a semantics-based protocol for real-time data access. In *Proceedings of the Real-Time Systems Symposium*, December 1993.

[3] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973. Reprinted in *Tutorial: Hard Real-Time Systems*, ed. J. Stankovic and K. Ramamritham, IEEE Computer Society 1988.

[4] C. D. Locke, D. R. Vogel, and T. J. Mester. Building a predictable avionics platform in ada: A case study. In *Proceedings of the Real-Time Systems Symposium*, pages 181–189, December 1991.

[5] Özalp Babaoğlu, K. Marzullo, and F. B. Schneider. A formalization of priority inversion. *Journal of Real-Time Systems*, 5:285–303, 1993.

[6] L. Sha, R. Rajkumar, S. H. Son, and C.-H. Chang. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793–800, July 1991.

[7] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

Table 1: Task set characteristics adapted from the generic avionics example in [4].

Task	Period (ms)	Exec. (ms)	Blocking* (ms)	Read set	Write set	Abort?
Timer_Interrupt	1	0.051	0	-	-	
Weapon_Release	200	3	9**	-	DB	
Radar_Tracking_Filter	25	2	9	DB,N	D,DB,T	Abort
RWR_Contact_Mgmt	25	5	9	DB,N,K,W	D,DB,T	Abort
Poll_Bus_Device	40	1	9	all	-	
Weapon_Aim	50	3	9	N,T	D,DB	
Radar_Target_Update	50	5	9	DB,N,K	D,DB,T	Abort
Nav_Update	59	8	9	DB,K,R	D,DB,T,R,W,RW	Abort
Display_Graphic	80	9	5	all	DB	Abort
Display_Hook_Update	80	2	5	DB	-	Abort
Tracking_Target_Upd	100	5	3	DB,N,K,R,RW	D,W	Abort
Weapon_Protocol	200	1	3	K	DB	
Nav_Steering_Cmds	200	3	3	D	D	
Display_Stores_Update	200	1	3	W	DB	
Display_Keyset	200	1	3	DB	all	
Display_Stat_Update	200	3	1	all	DB	Abort
BET_E_Status_Update	1000	1	1	-	D	
Nav_Status	1000	1	0	DB	D	

*Worst-case blocking based on treating whole tasks as transactions, as in [2] but in contrast to [4].

**Note that blocking for Weapon_Release as calculated in [4] can be no more than 1.745ms, despite conflict with Display_Graphic which has computation time 9ms and reads and writes DB. This difficulty is avoided in [2] by removing the jitter requirement for Weapon_Release.

Table 2: Schedulability calculations for the mixed locking/abort protocol.

Task	Period (ms)	Exec. (ms)	Blocking (ms)	Abort (ms)	Schedulability Test
Timer_Interrupt	1	0.051	0	0	$.051 < 1$
Weapon_Release	200	3.0505	1.0545	0	$\lceil 5/1 \rceil \times .051 + 3.0505 + 1.0545 = 4.36 < 5$
Radar_Tracking_Filter	25	2.0905	3.052	0.253	$\lceil 25/1 \rceil \times 0.051 + \lceil 25/200 \rceil \times 3.0505 + \lceil 25/25 \rceil \times 2.3435 + 3.052 = 9.721 < 25$
RWR_Contact_Mgmt	25	5.0935	3.052	5.578	$\lceil 25/1 \rceil \times 0.051 + \lceil 25/200 \rceil \times 3.0505 + \lceil 25/25 \rceil \times 2.3435 + \lceil 25/25 \rceil \times 10.6715 + 3.052 = 20.3925 < 25$
Poll_Bus_Device	40	1.0545	3.052	0	$\lceil 40/1 \rceil \times .051 + \lceil 40/200 \rceil \times 3.0505 + \lceil 40/25 \rceil \times (2.3435 + 10.6715) + \lceil 40/40 \rceil \times 1.0545 + 3.052 = 35.227 < 40$
Weapon_Aim	50	3.052	3.051	0	$\lceil 50/1 \rceil \times .051 + \lceil 50/200 \rceil \times 3.0505 + \lceil 50/25 \rceil \times (2.3435 + 10.6715) + \lceil 50/40 \rceil \times 1.0545 + \lceil 50/50 \rceil \times 3.052 + 3.051 = 36.7915 < 50$

Acknowledgments. This material is based upon work supported by the National Science Foundation under Grant No. CCR-9157629, with additional support from AT&T. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Window-Consistent Replication for Real-Time Applications *

Jennifer Rexford, Ashish Mehra, James Dolter and Farnam Jahanian

Real-Time Computing Laboratory
Department of EECS
The University of Michigan
Ann Arbor, MI 48109-2122

E-mail: {jrexford, ashish, jdolter, farnam}@eecs.umich.edu

Abstract

Two widely-studied approaches for structuring fault-tolerant services are the state-machine *and the* primary-backup *replication schemes. For a large class of soft and hard real-time applications, the degree of consistency among servers can be exploited to design replication protocols with* predictable *timing behavior. This is particularly useful in applications, such as automated process control, in which one can tradeoff the quality or precision for timely availability of data.*

This paper presents the architecture and prototype implementation of a primary-backup replication service that employs window consistency *semantics between the primary data repository and the backups. A client registers a data object with the service by declaring the consistency requirements for the data, in terms of a time* window. *The primary ensures that each backup site maintains a version of the object that was valid on the primary within the preceding time window by scheduling update messages to the backups.*

Decoupling the transmission of updates to the backups from the processing of client requests permits the primary to handle a higher rate of operations and provide more timely service to clients. The non-blocking semantics free the client from waiting for updates to the backups to complete. Furthermore, real-time scheduling of update messages can guarantee controlled inconsistency *between the primary and backup repositories.*

*The work reported in this paper was supported in part by the National Science Foundation under Grant MIP-9203895. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the NSF. Also supported by a grant from the Rackham School of Graduate Studies at the University of Michigan.

1 Introduction

A common approach to building fault-tolerant distributed systems is to replicate servers that fail independently. The objective is to give the clients the illusion of a service that is provided by a single server. Two widely-studied approaches for structuring fault-tolerant services are the *primary-backup* and the *state-machine* replication schemes. In both approaches, a *modify* request from a client results in the execution of an agreement protocol that ensures consistency among the replicated servers. For example, in the traditional primary-backup model, each client modify request to the primary repository requires an update transmission to the backup. This approach artificially ties the rate of client modify operations to the rate of updates to the backups, limiting both response time predictability and total system throughput while ensuring consistent data after fail-over. For a large class of soft and hard real-time applications, this restriction can be more detrimental than having a slightly stale copy of the data on the backups.

This paper presents the architecture of a new primary-backup scheme, referred to as the *window-consistent replication service* [1], which exploits the ability of many real-time applications to tolerate controlled time-inconsistency of the repository to provide timely availability of this data. The notion of window consistency relaxes atomic or causal consistency among replicas to obtain less expensive replication protocols. In particular, the proposed replication scheme exploits *temporal constraints* on objects to maintain a less current but acceptable version of the primary data on the backup. A client registers a data object with the service by declaring the consistency requirements for the data, in terms of a time *window*.

0-8186-5710-3/94 $3.00 © 1994 IEEE

The primary ensures that each backup site maintains a version of the object that was valid on the primary within the preceding time window by scheduling selective update messages to the backups. The time window essentially establishes a bounded distance between the primary and the backup states for each object. The following example illustrates the motivation for the proposed approach.

Example - Highly-Available Process Control System: Consider a primary-backup system for automated manufacturing and process control applications, as shown in Figure 1. The primary and the backup nodes share external devices such as sensors. The primary runs in a tight loop sampling sensors, calculating new values, and sending signal to external I/O under its control. The primary also maintains an in-memory data repository which is updated frequently during each iteration of the tight control-loop. One of the requirements on the system is to be able to switch to the backup in case of the primary failure within a few hundred milliseconds.

The in-memory data repository must be replicated on the backup to meet the strict timing constraint on the switch-over. Since there can be hundreds of updates to the data repository during each iteration of the control loop, it is impractical (and perhaps impossible) to update the backup synchronously each time the primary copy is changed. An alternative solution is to exploit the data semantics in a process control system by allowing the backup to maintain a less current but an acceptable copy of the data that resides on the primary. If the data on the backup does not fall too far behind the version on the primary, the backup can recover from a primary failure. For example, updates can be sent in batches or selectively to the backup. If the primary fails, the backup can take over even if the last few updates are lost. The objective is, however, to keep the backup data recent such that it can reconstruct a *consistent* system state by extrapolating from previous values and by reading new sensor values. However, one must ensure that the distance between the primary and the backup copies is bounded within a predefined time. In fact, different objects may have distinct tolerances in how far the backup can lag behind before the object state becomes stale. The challenge is to bound the distance between the primary and the backup such that consistency is not compromised while minimizing the overhead in exchanging messages between the primary and its backup. Under transient overload conditions, the system can gracefully degrade by allowing the backup to increase its distance from the primary. □

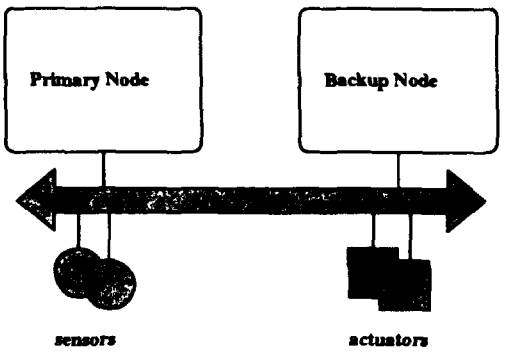


Figure 1: Primary-backup process control system

A replication scheme based on window consistency allows computations that may otherwise be disallowed by existing active or passive protocols that ensure atomic updates to a collection of replicas. Enforcing a weaker correctness criterion has been studied extensively for different purposes and application areas. In particular, a number of researchers have observed in the past that the notion of serializability is too strict a correctness criterion for real-time databases. Hence, several alternatives have been proposed that eliminate or relax serializability as a correctness criteria for managing consistency in real-time transactions. Among these are ϵ-serializability [2], similarity [3],temporal and external consistency [4], triggered real-time databases [5]. The above correctness criteria allow more concurrency by supporting a limited amount of inconsistency in how a transaction views the database state. The idea of imprecise computation is an interesting related approach that sacrifices accuracy for timeliness in real-time computations [6]. Exploiting weak consistency to obtain better performance has also been proposed in other non-real-time applications. For instance, in [7], the notion of quasi-copy is introduced which allows a weaker type of consistency between the central data and its cached copies at remote sites. The objective is to allow a cached copy to deviate from the central copy in a controlled way so that a scheduler has more flexibility in propagating updates. In the same spirit, the notion of window consistency provides controlled inconsistency among the replicas in a real-time application to support fault-tolerance.

2 Window Consistency

The window-consistent replication service consists of a primary and one or more backups, with the data

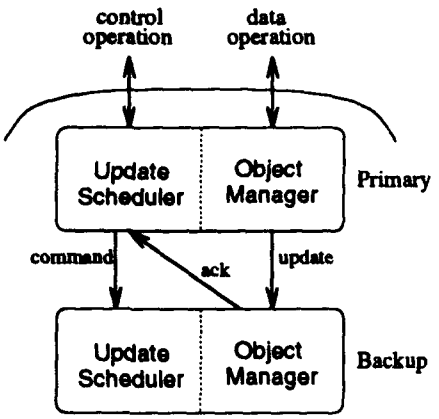


Figure 2: Window-consistent primary-backup architecture

on the primary shadowed at each backup. These sites store collections of objects which change over time, in response to client interaction with the primary. The primary handles client requests and ensures that each backup repository maintains a sufficiently recent version of the objects. In the absence of any failures, the primary satisfies all client requests and supplies a data-consistent repository. If the primary crashes, a window-consistent backup performs a fail-over to become the new primary by notifying the clients and other replicas of the new configuration. Service availability hinges on the existence of a window-consistent backup to replace a failed primary.

The service exports two sets of operations to the clients, namely, *control operations* and *data operations*, as shown in Figure 2. Control operations, such as object creation and deletion, are handled by the primary, and require complete agreement within the replication service. Client query and modify operations on the data are handled locally at the primary, without triggering interaction with the backups. The primary concurrently handles client requests and schedules selective updates to the backup repositories to guarantee the window consistency requirements of the data objects. The primary *object manager* (OM) satisfies client data requests, while sending update messages to the backup at the behest of the primary *update scheduler* (US).

Whenever the primary P modifies an object, P timestamps the new version; these timestamps identify successive versions of the object on the primary. The primary schedules transmissions to the backups to ensure that each backup has a sufficiently recent version of each object. At time t the primary has a copy of object O_i with timestamp $\tau_i^P(t)$, while a backup B stores a, possibly older, version $\tau_i^B(t)$. While B may have an older version of O_i than P, the copy on B must be "recent enough." The service must ensure that B always believes in data that was valid on P within the last δ_i time units. The value of δ_i is set during object creation, based on the time-consistency requirements of the application; P rejects the registration request if it cannot satisfy the object's window consistency requirements.

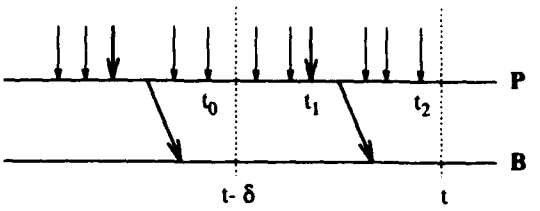


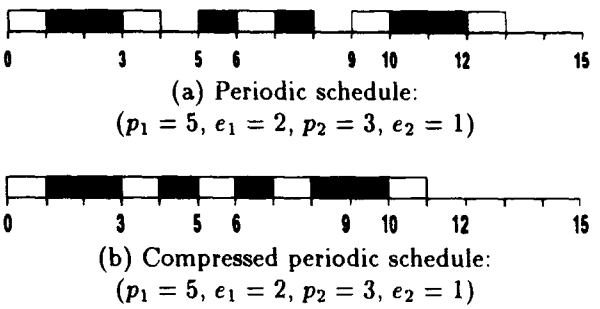
Figure 3: Window consistency semantics

B has a *window-consistent* copy of object O_i at time t if and only if $\tau_i^P(t-\delta_i) \leq \tau_i^B(t) \leq \tau_i^P(t)$. For example, in Figure 3, P performs several operations on O_i, on behalf of client requests, but selectively transmits update messages to B. At time t the primary has the most recent version of the object, which was timestamped at time t_2; the backup has a version first recorded on the primary at time t_1. Thus, $\tau_i^P(t) = t_2$, $\tau_i^B(t) = t_1$, and $\tau_i^P(t-\delta_i) = t_0$. Since $t_0 \leq t_1 \leq t_2$, B has a window-consistent version of O_i at time t. The primary US employs real-time scheduling algorithms to coordinate the selective updates for the various objects.

The primary P needs an accurate measure of the window consistency of the backup objects to determine the operating mode of the backup sites. Although P cannot know the exact value of τ_i^B at all times, the primary can estimate the state of the backup. In particular, P does know the time t_i^{xmit} when it last transmitted an update for O_i. The primary sent version $\tau_i^{xmit} = \tau_i^P(t_i^{xmit})$ of the object to the backup, so P knows $\tau_i^B \leq \tau_i^{xmit}$. However, if P has not received an acknowledgement for this update, P cannot guarantee that B has this copy of O_i yet. If the backups tell P what object versions they have received, then the primary knows the latest version τ_i^{ack} that P has seen acknowledged by B. The primary then knows that $\tau_i^{ack} \leq \tau_i^B$. Using τ_i^{xmit} and τ_i^{ack}, P can provide both optimistic and pessimistic measures of the window consistency of O_i on B.

A backup also needs some measure of its own window consistency to determine whether it is a suitable substitute for a failed primary. Since a window-

inconsistent backup B cannot supplant a crashed primary, B cannot tolerate long periods without hearing from P. The backups must balance the likelihood of false failure detection with the possibility of having no window-consistent backup to replace the crashed primary. Although B may be unaware of recent client interaction with P for each object, B does know τ_i^B and the time t_i^{xmit} when P transmitted this update. If less than δ_i time units have elapsed since P sent this update, then B has data that P believed within the last δ_i time units.



(a) Periodic schedule:
($p_1 = 5$, $e_1 = 2$, $p_2 = 3$, $e_2 = 1$)

(b) Compressed periodic schedule:
($p_1 = 5$, $e_1 = 2$, $p_2 = 3$, $e_2 = 1$)

Figure 4: Compressing a periodic schedule

3 Application of Real-Time Scheduling

The window-consistent replication model satisfies a high rate of client query and modify operations by relaxing the consistency constraints between the primary and the backups. Under the limitations of finite processing time and network bandwidth, however, the primary must *schedule* the selective updates to the backup sites. By casting the transmissions of updates as tasks, the primary US can draw upon real-time task scheduling algorithms. While several task models can accommodate window-consistent scheduling, we initially consider the periodic task model [8, 9].

With the periodic model, the primary coordinates transmissions to the backups by scheduling an update task with period p_i and service time e_i for each object O_i [1]. The end of a period serves as both the deadline for one invocation of the task and the arrival time for the subsequent invocation. The scheduler always runs the ready task with the highest priority, preempting execution if a higher-priority task arrives. Rate-monotonic scheduling statically assigns higher priority to tasks with shorter periods [8, 9], while earliest-due-date scheduling favors tasks with earlier deadlines [8].

Given a schedulable set of tasks, the primary US ensures that O_i is sent to the backup once per period p_i, resulting in a maximum time of $2p_i$ between successive transmissions of O_i. The replication service must consider object consistency requirements and transmission delays in determining p_i for each object. In the absence of a link failure, we assume a bound d on the end-to-end latency between the primary and the backups. If a client operation modifies O_i, the primary must send an update for the object within the next $\delta_i - d$ time units; otherwise, the backups may not receive a sufficiently recent version of O_i before the time-window δ_i elapses. For window consistency this permits a maximum period $p_i = (\delta_i - d)/2$.

Compressing the periodic schedule: While the periodic model can guarantee sufficient updates for each object, the schedule updates O_i only once per period p_i, even if computation and network resources permit more frequent transmissions. This restriction arises because the periodic model assumes that a task becomes ready to run only at period boundaries. However, the primary can transmit the current version of an object at any time. The scheduler can capitalize on this task readiness to improve both resource utilization and the window consistency on the backups by *compressing* the periodic schedule. Consider two objects O_1 and O_2 as depicted in Figure 4. The scheduler must send an update requiring 1 unit of processing time once every 3 time units (unshaded box) and an update requiring 2 units of processing time once every 5 time units (shaded box). For this example, both the rate-monotonic and earliest-due-date algorithms generate the schedule shown in Figure 4(a). While each update is sent as required in the major cycle of length 15, the schedule has 4 units of slack time. The periodic schedule can provide the *order* of task executions without restricting the *time* the tasks become active. If no tasks are ready to run, the scheduler can advance to the earliest pending task and activate that task by advancing the logical time to the start of the next period for that object. With the compressed schedule the primary still transmits an update for each O_i at least once per period p_i but can send more frequent updates when time allows. As shown in Figure 4(b), compressing the slack time allows the schedule to start over at time 11. In the worst case, the compressed schedule degrades to the standard periodic schedule with the associated guarantees. **Integrating a new**

[1] The size of O_i determines the time e_i required for each update transmission. In order to accommodate preemptive scheduling and objects of various sizes, the primary can send an update message as one or more fixed-length *packets*.

backup: To minimize the time the service operates without a window-consistent backup, the primary P needs an efficient mechanism to integrate a new or invalid backup. P must send the new backup B a copy of each object and then transition to the periodic schedule to sustain B. B must receive a copy of O_i in the "period" p_i before the periodic schedule begins; this ensures that B can afford to wait until the next p_i interval to start receiving periodic update messages for O_i. P guarantees a smooth transition to the periodic schedule by sending the objects to B in *reverse period* order, such that the objects with larger periods are sent before those with smaller periods. For object O_i, this ensures that only objects with smaller or equivalent periods can follow O_i in the integration schedule; these same objects can precede O_i in the periodic schedule. This guarantees that the integration schedule transmits O_i no more than p_i time units before the start of the periodic schedule, ensuring a consistent transition. The reverse-period integration schedule transmits a single copy of each object, minimizing the time required to establish window consistency on a new backup.

4 Prototype Implementation and Ongoing Work

We are developing a prototype implementation of the window-consistent replication service to demonstrate and evaluate the proposed service model. Each client or repository site is currently a Sun SPARCstation running Solaris 1.1. The sites communicate through UDP datagrams using the **Socket++** library from the University of Virginia, with extensions for *priority-based* access to the active sockets. The prototype implements rate-monotonic scheduling with compression. While Solaris 1.1 provides a stable environment for code development and testing, the platform does not support real-time thread scheduling, bounded communication delays, or synchronized clocks. After initial code development and testing, we will evaluate the service in a real-time distributed system [10].

The prototype provides a general framework for comparing the performance of different scheduling algorithms for coordinating update transmissions to the backups. We are currently investigating the distance-constrained task model [11] which assigns priorities based on separation restrictions. In addition, we are considering adaptive scheduling algorithms that incorporate knowledge of recent client interaction with the primary. These alternative models may permit more optimistic schedulability criteria (with some increased cost in scheduler complexity), allowing the replication service to accept objects with more demanding window consistency requirements.

Window consistency offers a framework for designing replication protocols with predictable timing behavior. By decoupling communication within the service from the handling of client requests, a replication protocol can handle a higher rate of query and modify operations and provide more timely response to clients. Scheduling the selective communication within the service provides bounds on the degree of inconsistency between servers. Although the current prototype implements the primary-backup replication model, we are exploring the application of window consistency to the state-machine approach to server replication. In addition, we are investigating the influence of controlled time-inconsistency on failure detection and recovery in replication protocols.

References

[1] A. Mehra, J. Rexford, J. Dolter, and F. Jahanian, "Window-consistent replication service," Technical report, Department of EECS, University of Michigan, April 1994.

[2] C. Pu and A. Leff, "Replica control in distributed systems: An asynchronous approach," in *Proc. ACM SIGMOD*, pp. 377–386, May 1991.

[3] T.-W. Kuo and A. K. Mok, "Application semantics and concurrency control of real-time data-intensive applications," in *Proc. Real-Time Systems Symposium*, 1992.

[4] K.-J. Lin, F. Jahanian, A. Jhingran, and C. D. Locke, "A model of hard real-time transaction systems," Technical Report RC 17515, IBM T.J. Watson Reseach Center, January 1992.

[5] H. F. Korth, N. Soparkar, and A. Silberschatz, "Triggered real time databases with consistency constraints," in *Proc. of the 16th VLDB Conference*, August 1990.

[6] J. Liu, K.-J. Lin, W.-K. Shih, R. Bettati, and J. Chung, "Imprecise computations," *to appear in IEEE Proceedings*, January 1994.

[7] R. Alonso, D. Barbara, and H. Garcia-Molina, "Data caching issues in an information retrieval system," *ACM Trans. Database Systems*, vol. 15, no. 3, pp. 359–384, September 1990.

[8] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, January 1973.

[9] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *Proc. Real-Time Systems Symposium*, pp. 166–171, December 1989.

[10] K. G. Shin, D. D. Kandlur, D. L. Kiskis, P. S. Dodd, H. A. Rosenberg, and A. Indiresan, "A distributed real-time operating system," *IEEE Software*, pp. 58–68, September 1992.

[11] C.-C. Han and K.-J. Lin, "Scheduling distance-constrained real-time tasks," in *Proc. Real-Time Systems Symposium*, pp. 300–308, December 1992.

Using Data Similarity to Achieve Synchronization for Free

Tei-Wei Kuo and Aloysius K. Mok
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712

Abstract

In [4], Graham proposes several conditions which are sufficient to guarantee that a transaction system will run serializably without any extra effort having to be taken. Systems satisfying these conditions are said to achieve serializability for free. The conditions considered by Graham are determined by a syntactic check on the transaction programs, and are independent of the semantics of data. In this paper, we use a semantic approach and propose a sufficient condition for achieving data synchronization for free which is based on the concept of data similarity [5]. Real-time transactions satisfying this condition can be scheduled correctly by any process scheduling discipline that is designed for the independent processes model [8], e.g., RMS, EDF, where no locking of data is assumed. The correctness of our approach is justified by exploiting the idea of Δ-serializability.

1 Introduction

There has been substantial interest in the profermance of transaction systems which have significant response time requirements. These requirements are usually posed as deadlines on individual transactions. A scheduling algorithm must attempt to meet deadlines as well as preserve database consistency. Due to priority inversion caused by data access, a transaction system with moderate utilization factor is often hard to schedule. In [5], we explored a weaker correctness criterion for concurrency control in real-time transactions, namely, the notion of "similarity", so that the schedulability problem is relaxed by the flexibility in scheduling read/write events introduced by the notion of similarity.

In [6], we proposed a class of real-time, data-access protocols called SSP (Similarity Stack Protocol) which is based on the notion of similarity. Transactions are allowed to run concurrently if their conflicting events are strongly similar. We gave a schedulability bound for SSP and also reported some encouraging simulation results. Although SSP was shown to compare favorably with Priority Ceiling Protocol (PCP) [11] and Stack Resource Policy (SRP) [2] especially on multiprocessor systems[1], a transaction system with a moderate workload is still hard to schedule. In this paper, we further explore the notion of similarity and propose a sufficient condition, so that a real-time transaction satisfying this condition can be scheduled correctly by an independent process scheduling algorithm such as RMS or EDF [8]. This means that the usually high utilization factor that can be achieved by these scheduling algorithms is also attainable for transactions satisfying our condition.

The rest of the paper is organized as follows. Section 2 summarizes the similarity concept and the correctness criterion in [5]. Section 3 describes a sufficient condition with which transactions can be scheduled independently. Section 4 further extends the results presented in section 3. Section 5 is the conclusion.

2 Data Similarity

2.1 Database Model

The state of a real-time system is represented by the values of a collection of data objects. Each data object takes its value from its *domain*. *Events* are primitive data operations (atomic read or write) which may occur many times in a computation. Each instance of an event is associated with a time stamp whose value is the (wall-clock) time at which the event instance occurs. A *transaction* is a partial order of events. An instance of a transaction is scheduled for every request

[0]Supported in part by a research grant from the Office of Naval Research under ONR contract number N00014-89-J-1472

[1]Notice that PCP and SRP are originally designed for single processor systems. Hence comparing them with SSP may not be germane.

0-8186-5710-3/94 $3.00 © 1994 IEEE

for the transaction. To distinguish between a transaction and an instance of it, we shall use the notation $\tau_{i,j}$ to denote the jth instance of transaction τ_i. The view of a transaction (instance) is a vector of data object values such that the ith component is the value read by the ith read event of the transaction instance [10].

A *schedule* over a set of transactions is a partial order of event instances issued by instances of the transaction set. Each event instance in a schedule is issued by one transaction instance. The ordering of event instances in a schedule must be consistent with the event ordering as specified by the transaction set. In a real-time computation, the partial ordering of event instances in a schedule is induced by the time stamps of event instances at different sites. A *serial schedule* is a sequence of transaction instances (i.e., a schedule in which the transaction instances are totally ordered).

2.2 Similarity

The value of a data object that models an entity in the real world cannot in general be updated continuously to perfectly track the dynamics of the real-world entity. The time needed to perform an update alone necessarily introduces a time delay which means that the value of a data object cannot be instantaneously the same as the corresponding real-world entity. Fortunately, it is often unnecessary for data values to be perfectly up-to-date or precise to be useful. In particular, data values of a data object that are slightly different are often interchangeable as read data for transactions. This observation underlies the concept of similarity among data values.

Similarity is a binary relation on the domain of a data object. Every similarity relation is reflexive and symmetric, but not necessarily transitive. Different transactions can have different similarity relations on the same data object domain. *Two views of a transaction are similar* iff every read event in both views uses similar values with respect to the transaction. We say that *two values of a data object are similar* if all transactions which may read them consider them as similar. In a schedule, we say that two *event instances are similar* if they are of the same type and access similar values of the same data object. We say that *two database states are similar* if the corresponding values of every data object in the two states are similar.

A minimal restriction on the similarity relation that makes it interesting for concurrency control is the requirement that it is preserved by every transaction, i.e., if a transaction T maps database state s to state t and state s' to t', then t and t' are similar if s and s' are similar. We say that a similarity relation is *regular* if it is preserved by all transactions. We are interested in regular similarity relations only.

2.3 Strong Similarity

The definition of regular similarity only requires a similarity relation to be preserved by every transaction, so that the input value of a transaction can be swapped with another in a schedule if the two values are related by a regular similarity relation. Unless a similarity relation is also transitive, in which case it is an equivalence relation, it is in general incorrect to swap events an arbitrary number of times in a schedule.

The notion of strong similarity was introduced in [5] which has the property that swapping similar events in a schedule will always preserve similarity in the output. This notion is motivated by the observation that the state information of many real-time systems is "volatile", i.e., they are designed in such a way that system state is determined completely by the history of the recent past, e.g., the velocity and acceleration of a vehicle are computed from the last several values of the vehicle's position from the position sensor. Unless events in a schedule may be swapped in such a way that a transaction reads a value that is derived from the composition of a long chain of transactions that extends way into the past, a suitable similarity relation may be chosen such that output similarity is preserved by limiting the "distance" between inputs that may be read by a transaction before and after swapping similar events in a schedule.

For the purpose of this paper, it suffices to note that if two events in a schedule are strongly similar (i.e., they are either both writes or both reads, and the two data values involved are strongly similar), then they can always be swapped in a schedule without violating data consistency requirements.

3 A Sufficient Condition

Similarity is an inherently application-dependent concept, and we expect the application engineer to define it for specific applications. In many real-time applications, it is often acceptable to use an older value of a sensor as input to a calculation, instead of waiting for a more up-to-date value. This is possible because the physics of the application may be such that changes in sensor reading over a short interval of time are so small as to be insignificant to the calculation. This

observation provides us with the needed connection between similarity and timing constraints governing data access.

Specifically, we assume that the application semantics allows us to derive a *similarity bound* for each data object such that two write events on the data object must be strongly similar if their time-stamps differ by an amount no greater than the similarity bound, i.e., all instances of write events on the same object that occur in any interval shorter than the similarity bound can be swapped in the (untimed) schedule without violating consistency requirements. Notice that the existence of a similarity bound does not imply that the similarity relation is transitive, since event swapping is based on (wall-clock) time values and not on the relative positions of events in a schedule.

3.1 Basic Idea

The basic idea is that transactions should not block one another as long as meeting timing constraints guarantees the strong similarity of their conflicting events. The event conflicts are resolved by appealing to the similarity bound in the following discussion which refers to Figure 1.

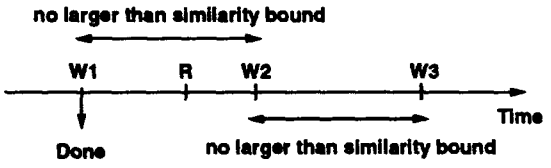


Figure 1: similarity of conflicting events

Suppose two events e_1 and e_2 conflict with each other. Let e_1 and e_2 be the write events w_2 and w_3, respectively. If their write values are similar under the similarity bound as shown in Figure 1, these two write events are strongly similar and it does not matter which write value is read by subsequent read events. Suppose e_1 and e_2 are respectively, the write event w_2 and the read event r in Figure 1. For their relative ordering to be unimportant, there must exist an earlier write event whose write value is similar to the write value of w_2 under the similarity bound. If this is the case, as is shown in Figure 1, then it does not matter which write value the read event r reads. The same argument applies to the case where e_1 and e_2 are a read event and a write event, respectively.

3.2 A Sufficient Condition: Timing Constraints vs Data Similarity

Suppose sb_i is a similarity bound for a data object x_i. Any two writes on x_i within an interval shorter than sb_i are interchangeable because they are strongly similar. Let p_i^{max}, p_i^{nxt}, and p_i^{min} be the maximum, the second largest, and the minimum periods of transactions updating x_i, respectively. If there is only one transaction updating x_i, then p_i^{max} is equal to p_i^{min} and p_i^{nxt}. Suppose p_i^r is the maximum period of transactions reading x_i. In the following, we shall derive a sufficient condition which guarantees the "strong similarity" of any concurrently executing transaction instances.

For simplicity of discussion, we assume in this paper that the deadline of a transaction instance is equal to the end of its period. Extension of our results to relax this restriction is straightforward.

Write vs Write Condition: $(p_i^{max} + p_i^{nxt}) \leq sb_i$

By our definition of strong similarity, two conflicting write events are interchangeable if they are strongly similar. In order words, conflicting write events of any overlapping transaction instances are interchangeable if these write events are strongly similar. (We say that two transaction instances overlap if their execution overlap in time.) If no transaction misses its deadline, the maximum temporal distance between any two conflicting write events of overlapping transaction instances on data object x_i is $(p_i^{max}+p_i^{nxt})$. Obviously, if $(p_i^{max} + p_i^{nxt}) \leq sb_i$, conflicting write events of any overlapping transaction instances are strongly similar and interchangeable.

Notice that the Write vs Write condition for data object x_i can be ignored if there is only one transaction updating x_i. This is because no two instances of the same transaction will overlap if the transaction never misses its deadline. □

Read vs Write Condition: $(p_i^{max} + 2p_i^{min} + p_i^r) \leq sb_i$

Suppose τ is a transaction with period p_i^r and reads data object x_i. To ensure correctness, conflicting write events which might be read by an instance of τ must be strongly similar (thus interchangeable) so that any instance of τ will not block or be blocked by transaction instances which may update x_i.

If no transaction updating x_i misses its deadline, then no read event e_r can read from a conflicting write event which occurs more than $2p_i^{min}$ ago. Let this oldest write event be called $write^{old}$ of e_r. For ease of argument, we assume without loss of general-

ity that the initial database state is determined by a fictitious set of write events so that an oldest write event always exists. On the other hand, a transaction instance which overlaps with the transaction instance issuing e_r may issue a conflicting write event almost p_i^{max} later than the end of the period of the transaction instance issuing e_r. Let this write event be $write^{young}$ of e_r. Obviously, this transaction instance of τ (which issues e_r) should not block or be blocked by any transaction instance because of read-write access conflict on x_i, assuming that the maximum temporal distance of $write^{old}$ and $write^{young}$ of e_r is no more than the similarity bound sb_i of x_i. In order words, read-write access conflict of x_i can be resolved if $(p_i^{max} + 2p_i^{min} + p_i^r) \leq sb_i$.

When there is only one transaction updating x_i, then $p_i^{max} = p_i^{min}$. □ (In the last case, further optimization is possible.)

We claim that, if a transaction set satisfies the *Read vs Write* and *Write vs Write* conditions, then these transactions can be scheduled independently as if they do not share data (with the usual assumption that individual read and write events are atomic). Formal justification of this claim is stated in Theorem 2 below.

Suppose two schedules π and π' have the same event set E, Δ and $\Delta^{\#}$ are respectively a strong similarity relation and a regular similarity relation for both π and π'. We say that π' is a *derived* schedule of π if for any read event that appears in π and π', the two corresponding write events in π and π' read by the read event are strongly similar in π, and the last write events which update the same data object in π and π' are strongly similar in π.

Theorem 1 *[5, 7] Suppose two schedules π and π' have the same event set E, and Δ, $\Delta^{\#}$ are, respectively, a strong similarity relation and a regular similarity relation for both π and π'. If π' is a derived schedule of π, then π and π' are view-similar under $\Delta^{\#}$, i.e., π and π' transform similar states (under Δ) into similar states (under $\Delta^{\#}$).*

Notice that view-similarity is an extension of view equivalence [5, 10]. A schedule is view Δ-serializable if it is view similar to a serial schedule.

Theorem 2 *If a transaction set satisfies both the Read vs Write and Write vs Write conditions, then any schedule that satisfies all transaction deadlines is view Δ-serializable.*

Proof. The proof follows directly from Theorem 1 if there exists a serial schedule π' which is a derived schedule of any schedule π that satisfies all transaction deadlines. According to the definition of "derived schedule", π and π' must satisfy the following two requirements: (1) all write events read by the same read event in π and π' must be strongly similar in π, and (2) the last write events on every data object in π and π' must be strongly similar in π. In the following, we shall prove that there exists a sequence of event swaps from π to some serial schedule π' such that the requirements of a derived schedule are preserved at every step in the sequence.

Since any conflicting write events of overlapping transaction instances in π are strongly similar (according to the *Write vs Write* condition), they can be swapped in any way without violating the second requirement of a derived schedule. Likewise, a conflicting read event and a conflicting write event of two overlapped executing transaction instances in π can be swapped in any way without violating the first requirement of a derived schedule, because they are "strongly similar" according to the *Read vs Write* condition. In particular, instances of all write events on the same data object that occur in any interval shorter than the similarity bound can be swapped in a (untimed) schedule without violating consistency requirements. Thus, swapping such write events will not violate the first requirement of a derived schedule. Therefore, conflicting events of overlapping transaction instances can be swapped in any order. Since non-conflicting events can also be swapped in any order, events of overlapping transaction instances can be swapped in any order. In other words, overlapping transaction instances can be serialized in any order. Also, transaction instances which are not overlapped in π are already serialized. Therefore, π can be serialized by swapping events of overlapping transaction instances in any order. □

3.3 Extensions

Since different transactions may have different precision requirements for a data object, the *Read vs Write* and *Write vs Write* conditions can be weakened. Suppose sb_i' is the similarity bound of a data object x_i with respect to a transaction τ'. The *Read vs Write* condition can be weaken to: $(p_i^{max} + 2p_i^{min} + p') \leq sb_i'$ if the period of τ is p'. The *Write vs Write* condition can be weakened to: $(p_i^{max} + p_i^{nxt}) \leq sb_i'$.

Finally, we consider the situation where some transactions satisfy the *Read vs Write* and *Write vs Write* conditions, but others do not. In this case, the transaction system cannot be scheduled "fully" independently. A simple variation of Similarity Stack Protocol (SSP) [6] can be made to take care of this situation,

as follows.

As in SSP, transactions are partitioned into interactive sets such that no two transactions in different interactive sets may share any data object. If all transactions in an interactive set satisfy the *Read vs Write* and *Write vs Write* conditions, the recency bound of the interactive set can be set to ∞ such that transactions in the interactive set can be scheduled independently of one another. Here, the recency bound of an interactive set limits the length of any interval spanned by overlapping transaction instances in the set. If any transaction in an interactive set fails any one of the conditions, the recency bound of the interactive set is calculated as defined in [6]. The correctness of this approach can be justified by an argument similar to the last section.

4 Conclusion and Future Research

In [4], Graham proposes several conditions which are sufficient to guarantee that a transaction system will run serializably without any extra effort having to be taken. Systems satisfying these conditions are said to achieve serializability *for free*. The conditions considered by Graham are determined by a syntactic check on the transaction programs, and are independent of the semantics of data. In this paper, we take a semantic approach and propose a sufficient condition for achieving data synchronization for free which is based on the concept of *data similarity* [5]. Real-time transactions satisfying this condition can be scheduled correctly by any process scheduling discipline that is designed for the independent processes model [8] (e.g., RMS, EDF) where no locking of data is assumed. With our approach, the usually high utilization factor that can be achieved by these scheduling disciplines is also attainable for transactions satisfying our condition.

We believe that there are many interesting research issues concerning the concept of similarity. To gain experience, it is important to investigate how to construct similarity relations systematically from application specifications. A toolset which facilitates reasoning about similarity relations for typical real-time applications should be very useful.

References

[1] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *Proceeding of the 14th VLDB Conference*, Los Angeles, CA 1988, pp. 1-12.

[2] T.P. Baker, "A Stack-Based Resource Allocation Policy for Real Time Processes," *IEEE 11th Real-Time Systems Symposium*, December 4-7, 1990.

[3] S. B. Davidson and A. Watters, "Partial Computation in Real-Time Database Systems," *The 5th Workshop on Real-time Software and Operating Systems*, May 1988, pp. 117-121.

[4] Marc H. Graham, "How to Get Serializability for Real-Time Transactions without Having to Pay for It," *IEEE 14th Real-Time Systems Symposium*, December 1993.

[5] Tei-Wei Kuo and Aloysius K. Mok, "Application Semantics and Concurrency Control of Real-Time Data-Intensive Applications", *IEEE 13th Real-Time Systems Symposium*, 1992.

[6] Tei-Wei Kuo and Aloysius K. Mok, "SSP: a Semantics-Based Protocol for Real-Time Data Access," *IEEE 14th Real-Time Systems Symposium*, December 1993.

[7] Tei-Wei Kuo, "Real-Time Database — Semantics and Resource Scheduling," Ph.D. dissertation, University of Texas at Austin, 1994.

[8] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a hard real-time environment," *Journal of the ACM*, Vol. 20 No. 1, January 1973, pp. 46-61.

[9] Kwei-Jay Lin, Swami Natarajan, and Jane W.-S. Liu, "Imprecise Results: Utilizing Partial Computations in Real-Time Systems," *IEEE 8th Real-Time Systems Symposium*, December 1987, pp. 210-217.

[10] C. Papadimitriou, "The Theory of Database Concurrency Control," *Computer Science Press*, 1986.

[11] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols : An Approach to Real-Time Synchronization," *Technical Report* CMU-CS-87-181, Dept. of Computer Science, CMU, November, 1987. *IEEE Transactions on Computers*, Vol. 39, No. 9, September 1990.

Index of Authors

IEEE Computer Society

IEEE Computer Society Press Publications

Monographs: A monograph is an authored book consisting of 100-percent original material.
Tutorials: A tutorial is a collection of original materials prepared by the editors and reprints of the best articles published in a subject area. Tutorials must contain at least five percent of original material (although we recommend 15 to 20 percent of original material).
Reprint collections: A reprint collection contains reprints (divided into sections) with a preface, table of contents, and section introductions discussing the reprints and why they were selected. Collections contain less than five percent of original material.
Technology series: Each technology series is a brief reprint collection — approximately 126-136 pages and containing 12 to 13 papers, each paper focusing on a subset of a specific discipline, such as networks, architecture, software, or robotics.
Submission of proposals: For guidelines on preparing CS Press books, write the Managing Editor, IEEE Computer Society Press, PO Box 3014, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-1264, or telephone (714) 821-8380.

Purpose

The IEEE Computer Society advances the theory and practice of computer science and engineering, promotes the exchange of technical information among 100,000 members worldwide, and provides a wide range of services to members and nonmembers.

Membership

All members receive the acclaimed monthly magazine *Computer*, discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others seriously interested in the computer field.

Publications and Activities

***Computer* magazine:** An authoritative, easy-to-read magazine containing tutorials and in-depth articles on topics across the computer field, plus news, conference reports, book reviews, calendars, calls for papers, interviews, and new products.
Periodicals: The society publishes six magazines and five research transactions. For more details, refer to our membership application or request information as noted above.
Conference proceedings, tutorial texts, and standards documents: The IEEE Computer Society Press publishes more than 100 titles every year.
Standards working groups: Over 100 of these groups produce IEEE standards used throughout the industrial world.
Technical committees: Over 30 TCs publish newsletters, provide interaction with peers in specialty areas, and directly influence standards, conferences, and education.
Conferences/Education: The society holds about 100 conferences each year and sponsors many educational activities, including computing science accreditation.
Chapters: Regular and student chapters worldwide provide the opportunity to interact with colleagues, hear technical experts, and serve the local professional community.